



СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
SIBIRIAN FEDERAL UNIVERSITY

Е. Д. Карпова

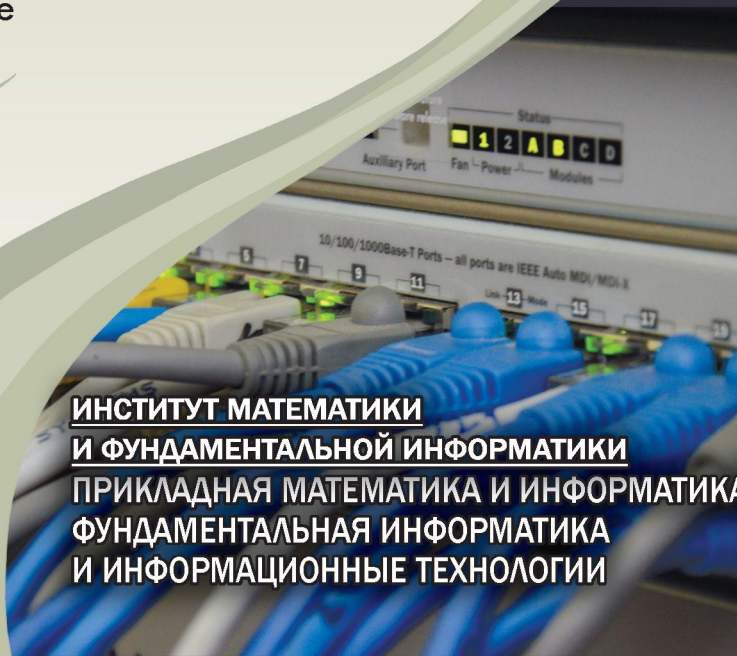
ОСНОВЫ МНОГОПОТОЧНОГО И ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Учебное
пособие

УМО



ИНСТИТУТ МАТЕМАТИКИ
И ФУНДАМЕНТАЛЬНОЙ ИНФОРМАТИКИ
ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА
ФУНДАМЕНТАЛЬНАЯ ИНФОРМАТИКА
И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ



Е. Д. Карпова

ОСНОВЫ МНОГОПОТОЧНОГО
И ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Министерство образования и науки Российской Федерации
Сибирский федеральный университет

Федеральное государственное бюджетное учреждение науки
«Институт вычислительного моделирования
Сибирского отделения Российской академии наук»

Сибирский научно-образовательный центр
суперкомпьютерных технологий

Е. Д. Карпова

ОСНОВЫ МНОГОПОТОЧНОГО И ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Допущено УМО по классическому университетскому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлениям ВПО 010400 «Прикладная математика и информатика» и 010300 «Фундаментальная информатика и информационные технологии», 23 марта 2015 г.

Красноярск
СФУ
2016

УДК 004.272(07)
ББК 32.973я73
К225

Карпова, Е. Д.
К225 Основы многопоточного и параллельного программирования :
учеб. пособие / Е. Д. Карпова. – Красноярск : Сиб. федер. ун-т, 2016. –
356 с.
ISBN 978-5-7638-3385-0

Рассматриваются современные подходы к разработке программного обеспечения для высокопроизводительных параллельных вычислительных систем. Приводятся общие сведения об архитектурах современных суперкомпьютеров и методах их программирования. Описываются особенности ряда популярных средств разработки многопоточных и параллельных программ и их использования для эффективного решения научных и прикладных задач.

Предназначено для студентов, аспирантов, инженеров и исследователей, работающих в области прикладной математики, вычислительной физики и высокопроизводительных параллельных вычислений.

*Работа выполнена при частичной поддержке Российского научного фонда
(проект № 14-11-00147)*

Электронный вариант издания см.:
<http://catalog.sfu-kras.ru>

УДК 004.272(07)
ББК 32.973я73

ISBN 978-5-7638-3385-0

© Сибирский федеральный
университет, 2016

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ.....	5
ВВЕДЕНИЕ.....	8
Г л а в а 1. ОБЗОР ОБЛАСТИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ	10
1.1. Основные архитектурные особенности построения параллельной вычислительной среды	10
1.2. Основные классы современных параллельных компьютеров	17
1.3. Разработка параллельных приложений	25
1.4. Программные средства.....	29
1.5. Парадигмы параллельных приложений	34
Контрольные вопросы и задания	51
Задачи	52
Г л а в а 2. ПРОБЛЕМЫ ПРОГРАММИРОВАНИЯ ДЛЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ С ОБЩЕЙ ПАМЯТЬЮ	54
2.1. Анатомия потока	54
2.2. Синхронизация потоков. Оператор ожидания.....	58
2.3. Взаимное исключение. Задача критической секции	60
2.4. Сигнализирующие события. Барьерная синхронизация.....	65
2.5. Семафоры.....	73
2.6. Мониторы	85
Контрольные вопросы и задания	91
Задачи	93
Г л а в а 3. УПРАВЛЕНИЕ ПОТОКАМИ С ПОМОЩЬЮ ФУНКЦИЙ WinAPI	95
3.1. Объекты ядра.....	95
3.2. Процессы.....	101
3.3. Потоки	108
3.4. Синхронизация потоков в пользовательском режиме ...	113
3.5. Синхронизация потоков с помощью объектов ядра	120
3.6. Проблемы условной синхронизации	128
3.7. Проецируемые в память файлы.....	138
3.8. Совместный доступ процессов к данным через механизм проецирования	144
Контрольные вопросы и задания	146
Задачи	147

Г л а в а 4. ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ OpenMP	153
4.1. Программная модель OpenMP	153
4.2. Модель памяти OpenMP	156
4.3. Среда выполнения OpenMP-программы	158
4.4. Директива <code>omp parallel</code>	164
4.5. Распределение работы в параллельной области по нитям	171
4.6. Директивы синхронизации	191
4.7. Переменные среды и функции времени выполнения	204
4.8. Спецификации стандарта OpenMP v. 4.0	210
4.9. Оптимизация программ OpenMP	212
4.10. Ограничения OpenMP	213
Контрольные вопросы и задания	215
Задачи	215
Г л а в а 5. СОГЛАСОВАННОЕ ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ	221
5.1. Проблемы программирования для вычислительных систем с распределенной памятью	221
5.2. Оценка эффективности параллельных алгоритмов	225
5.3. Реализация базовых алгоритмов вычислительной математики	248
5.4. Проблемы выбора эффективной параллельной реализации	272
Контрольные вопросы и задания	289
Г л а в а 6. ОСНОВЫ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ MPI	291
6.1. Архитектурная парадигма MPI	291
6.2. Обрамляющие и информационные функции MPI	293
6.3. MPI и крупноблочное распараллеливание	294
6.4. Организация вычислений	300
6.5. Организация взаимодействий процессов	311
6.6. Повышение эффективности MPI-программ	335
Контрольные вопросы и задания	338
Задачи	340
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	347

ПРЕДИСЛОВИЕ

Предлагаемое учебное пособие содержит материал для базового курса «Параллельное программирование», который охватывает широкий круг вопросов, связанных с высокопроизводительными вычислениями, а также отражает содержание нескольких спецкурсов, которые на протяжении ряда лет читаются автором в Институте математики и фундаментальной информатики Сибирского федерального университета и аспирантуре Института вычислительного моделирования СО РАН.

Пособие можно разделить на три части: общие аспекты параллельного программирования; программирование для параллельных вычислительных систем (ПВС) с общей памятью; программирование для систем с распределенной памятью. В каждой части сначала обсуждаются проблемы, порождаемые архитектурой соответствующей вычислительной системы (ВС), теоретические основы решения этих проблем, затем подробно рассматривается языковая реализация всех базовых механизмов. Как теоретический, так и практический материал проиллюстрирован большим количеством примеров, многие из которых давно уже стали классикой параллельного программирования.

В *первой главе* рассматриваются основные особенности параллельного программирования. Приводится краткая классификация основных видов архитектур ПВС, и подчеркивается необходимость использования разнообразных методов написания параллельных программ в зависимости от типа архитектуры. Здесь же дан краткий перечень существующих программных средств, методов написания параллельных программ, а также факторов, определяющих те или иные приемы распараллеливания.

Во *второй главе* более подробно описываются особенности разработки программ для ПВС с общей памятью. Дается определение потока, рассматриваются основные состояния, в которых может находиться поток. Приводятся сведения о создании параллельных потоков и методах решения проблем, порождаемых необходимостью их синхронизации (взаимного исключения, барьерной синхронизации, условного ожидания), вводятся основные синхропримитивы, используемые при многопоточном программировании. Подробно рассмотрено использование семафоров и мониторов. Обсуждаются способы решения классических задач многопоточного программирования (задачи о критической секции, кольцевом буфере, философах, читателях и писателях).

Третья глава посвящена практическому применению потоков для операционной системы Windows с использованием WinAPI. Дается общее понятие объекта ядра ОС Windows, процесса и потока. Приводится описание общей структуры создаваемой многопоточной программы. Обсуждаются особенности реализации проблемы взаимного исключения потоков одного процесса, т. е. потоков, находящихся в общем адресном пространстве (синхронизация в пользовательском режиме), и общий случай синхронизации потоков, относящихся, может быть, к разным процессам, с помощью объектов ядра. Рассмотрен один из простых механизмов организации связи между потоками разных процессов, например, для обмена данными. Автор благодарит аспиранта Г. А. Федорова за помощь в отборе материала и отладке кода ряда примеров.

Технология OpenMP создания параллельных программ для ВС с общей памятью обсуждается в *четвертой главе*. В отличие от реализации многопоточности в языке Си с помощью функций WinAPI, или библиотек Pthread, или Qt библиотека OpenMP в большей степени рассчитана на прикладного программиста, позволяя быстро создавать короткие и простые многопоточные приложения с помощью директив компилятора из последовательного кода. Знание основ технологии OpenMP полезно еще и по той причине, что ее современные реализации обеспечивают поддержку программирования для комбинированных ПВС, содержащих как общую, так и распределенную память.

Общие сведения о разработке параллельных программ для систем с распределенной памятью на основе механизма передачи сообщений приводятся в *пятой главе*. Кратко обсуждаются вопросы, связанные с исследованием информационных зависимостей и оценками внутреннего параллелизма алгоритмов, даются понятия ускорения, эффективности, масштабируемости. На ряде примеров показаны различные стратегии использования вычислительных ресурсов ПВС, взаимодействующих между собой через механизм передачи сообщений. Рассмотрены достоинства и недостатки каждого из подходов. Описаны двухточечные и коллективные взаимодействия процессов, подходы к оценке времени, необходимого на передачу сообщений в кластерных системах. В конце главы рассмотрены основные задачи вычислительной математики и схемы возможных параллельных алгоритмов для их решения.

В *шестой главе* рассмотрены основы программирования для ВС с распределенной памятью с помощью библиотеки MPI, которая соответствует всем требованиям одноименного стандарта средств организации передачи сообщений (message passing interface – MPI). Описана архитектурная парадигма MPI, ее связь с крупноблочным распараллеливанием. Обсуждаются вопросы организации вычислений (использование комму-

никаторов, производных типов, виртуальных топологий) и взаимодействий процессов (двухточечные и коллективные обмены, операции приведения и барьерной синхронизации). Отметим, что хотя в тексте и приведены синтаксис и характеристика большинства функций MPI для языка Си, главу не следует рассматривать как описание стандарта MPI. Все вводимые концепции, понятия и методы проиллюстрированы примерами. Автор благодарит А. В. Малышева за предоставленный материал и полезные обсуждения по теме главы.

Знания и навыки, полученные при изучении курса, позволяют в дальнейшем перейти к более детальному освоению инструментальных средств разработки параллельных программ и методов эффективного распараллеливания практических и научных задач.

Базовый курс «Параллельное программирование» читается в Институте математики и фундаментальной информатики Сибирского федерального университета в течение восьми лет. В это же время автором читались и более узконаправленные спецкурсы и курсы для магистрантов и аспирантов, материалы которых также включены в пособие.

Следует отметить, что при разработке курса и подготовке настоящего учебного пособия автором использовались, прежде всего, учебные пособия и монографии [1, 2, 9–14, 25–27, 34, 35, 48], материалы по параллельному программированию, представленные на порталах [77, 82], а также электронный курс [68] «Многопроцессорные вычислительные системы и параллельное программирование», разработанный под руководством профессора В. П. Гергеля в Нижегородском госуниверситете им. Н. И. Лобачевского.

Автор благодарит за полезные обсуждения члена-корреспондента РАН В. В. Шайдурова, профессора А. В. Старченко, профессора А. И. Легалова, Д. А. Кузьмина, а также признателен за помощь и участие Андрею Малышеву, Юрию Шанько, Георгию Федорову, Екатерине Дементьевой и Марине Вдовенко.

Работа выполнена при частичной поддержке Российского научного фонда (проект № 14-11-00147).

ВВЕДЕНИЕ

Сегодня многопоточное и параллельное программирование применяется не только для научных вычислений, но и в повседневных областях человеческой деятельности. Это обуславливается массовым переходом производителей микропроцессоров на многоядерные архитектуры. Любой современный персональный компьютер является параллельной вычислительной системой, а многие приложения – суть многопроцессные (или многопоточные). Многие научные и промышленные предприятия используют для повседневных нужд высокопроизводительные вычислительные системы, состоящие из десятков тысяч процессорных узлов.

Очевидно, что при разработке параллельных программ необходимо применять методы, обеспечивающие эффективное использование предоставляемых вычислительных ресурсов. Однако разнообразие архитектур ПВС привело к тому, что в настоящее время не существует языка, позволяющего создавать программы, легко и эффективно переносимые с одной архитектуры на другую.

Первоначально виделось, что использование языков последовательного программирования обеспечит универсальный путь для написания параллельных приложений. Однако быстро выяснилось, что такой подход обладает рядом недостатков:

- Прямое распараллеливание последовательных программ часто не обеспечивает достижения приемлемого уровня параллелизма из-за ограничений, обусловленных принципиально последовательными реализациями алгоритмов и стереотипами последовательного мышления.

- В большинстве языковых средств для реализации параллелизма необходимо явно формировать все параллельные фрагменты и следить за корректной синхронизацией процессов.

- Допустимость использования в языках «ручного» управления памятью может привести к конфликтам между процессами в борьбе за общий ресурс.

- Разработанные программы оказываются жестко привязанными к конкретной архитектуре или к семейству архитектур (дальнейшая смена поколений вычислительных систем или появление новых, более эффективных, архитектур ведут к потере разработанного ПО и/или необходимости его переработки, что не раз случалось на различных этапах развития программирования).

- Существует излишняя детализация ведения вычислений, поскольку кроме управления, связанного с непосредственным преобразованием дан-

ных, необходимо явно прописывать механизмы безопасного и корректного использования общих данных с учетом специфических особенностей конкретной архитектуры ВС.

- Создание ПВС с архитектурой смешанного типа требует комплексного использования комбинированных методов явного описания параллелизма решаемой задачи, что, в свою очередь, еще больше усложняет параллельное программирование.

Несмотря на все перечисленные трудности, в настоящее время существует большое количество средств и методов разработки специализированных многопоточных, распределенных и параллельных приложений, что в итоге позволяет решать важные научные и практические задачи. Постоянный рост числа таких задач и удешевление высокопроизводительных вычислительных систем ведут к тому, что разработка архитектурно зависимых программ становится экономически выгодной. Поэтому современный квалифицированный программист должен знать разнообразные методы параллельного программирования и уметь их использовать для построения эффективных приложений. Большинство современных учебных программ подготовки специалистов в таких областях, как прикладная математика и компьютерные науки, информатика и вычислительная техника, предполагают некоторое количество курсов, связанных с проблемами высокопроизводительных вычислений. Отметим, что настоящее учебное пособие включает довольно современный материал, достаточно быстро ставший классическим для быстро развивающейся области современного параллельного программирования.

Глава 1 | ОБЗОР ОБЛАСТИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

В главе обсуждается широкий круг тем, связанных с параллельным программированием. Рассмотрены принципы, положенные в основу параллельной обработки данных, проблемы классификации современных суперкомпьютеров, особенности их архитектуры. Изложены вопросы, связанные с проектированием параллельных приложений, описаны и проиллюстрированы на классических примерах их главные парадигмы.

1.1. Основные архитектурные особенности построения параллельной вычислительной среды

Следуя [9], выделим два основных фактора, определяющих бурное развитие вычислительной техники на пути увеличения производительности вычислительных систем, уменьшения их размеров, а также расширения круга решаемых задач: 1) развитие элементной базы; 2) совершенствование архитектуры.

Сравним характеристики (табл. 1.1) одного из первых компьютеров мира – EDSAC, появившегося в 1949 году в Кембридже, – и одного процессора современного суперкомпьютера Roadrunner¹, установленного в 2008 году в Лос-Аламосской национальной лаборатории (США).

За 60 лет тактовая частота процессора выросла в тысячи раз, а производительность увеличилась в сотни миллионов. Ясно, что основное увеличение производительности обусловлено не наращиванием мощности процессора, а использованием новых решений в архитектуре компьютеров.

Архитектуру однопроцессорного компьютера принято называть архитектурой фон Неймана (рис. 1.1). Она была логичным решением² проблемы создания первой электронной машины с запоминаемой программой

¹ Суперкомпьютер Roadrunner создан компанией IBM для Министерства энергетики США и установлен в Лос-Аламосской национальной лаборатории в Нью-Мексико, США. Roadrunner первым в мире преодолел на тесте Linpack рубеж производительности в 1 PFlop/s.

² В 1948–1950 гг. в Советском Союзе независимо от Джона фон Неймана под руководством Сергея Александровича Лебедева была разработана и реализована МЭСМ (малая электронная счетная машина), основанная на сходных идейных и архитектурных принципах.

и показала свою жизнеспособность. Основными компонентами для выполнения программ в такой архитектуре являются центральное процессорное устройство (ЦПУ), кэш- и оперативная память. Процессор выбирает инструкции из памяти, декодирует их и выполняет. Он содержит управляющее устройство (УУ), арифметико-логическое устройство (АЛУ) и регистры. УУ вырабатывает сигналы, управляющие действиями АЛУ, системой памяти и внешними устройствами. АЛУ выполняет арифметические и логические инструкции, определяемые набором инструкций процессора. В регистрах хранятся инструкции, данные и состояние машины, в том числе счетчик команд.

Таблица 1.1

Характеристика	EDSAC (1949)	1 процессор Roadrunner (2008)	Значение увеличилось
Тактовая частота	0,5 МГц	3200 МГц	↑ 6,4e+3
Пиковая производительность ¹	1,0e+2 оп/с	1,28e+10 флопс	↑ 1,28e+8

Однако архитектура однопроцессорного компьютера начала изменяться и совершенствоваться буквально с первых шагов развития вычислительной техники, что, прежде всего, связано с идеями параллелизма.

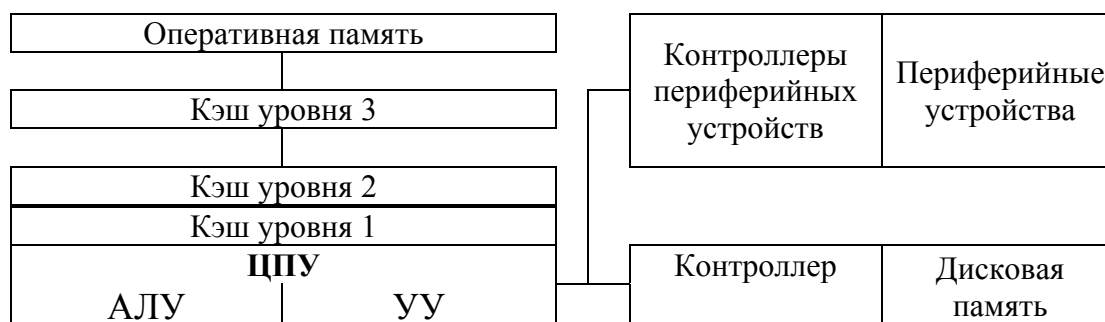


Рис. 1.1. Принципиальная схема однопроцессорной ЭВМ

Возможные пути достижения параллелизма детально рассматриваются в монографиях [8, 16, 32, 39, 43, 48, 51, 56, 64, 72], в этих же работах описывается история развития параллельных вычислений и приводятся примеры конкретных параллельных ЭВМ.

¹ При подсчете пиковой производительности предполагается, что все устройства компьютера работают в максимально производительном режиме. Пиковая производительность – величина теоретическая, и практически никогда не достигается. Измеряется, как правило, в следующих единицах: 1) число команд в единицу времени – MIPS (Million Instructions Per Seconds) (недостаток такого сравнения очевиден – команды процессора разные для разных машин); 2) число операций (сложения) для чисел с плавающей точкой в единицу времени – Flops (Floating Points Operations Per Seconds).

Параллелизм опирается на следующие архитектурные особенности построения вычислительной среды.

1. *Независимость функционирования отдельных устройств ЭВМ* – в настоящее время утверждение справедливо для всех основных компонентов ВС: для устройств ввода-вывода, для обрабатывающих процессоров и для устройств памяти.

2. *Иерархическое устройство памяти* – очень важная особенность архитектуры, влияющая на его производительность (рис. 1.2, табл. 1.2).



Рис. 1.2. Иерархия памяти в современном компьютере

Таблица 1.2

Вид памяти	Размер	Скорость доступа
Регистры	Несколько десятков байт	1 такт работы
Кэш первого уровня	Несколько десятков килобайт	1–2 такт работы
Кэш второго уровня	Несколько мегабайт	8–20 тактов работы
Кэш третьего уровня	До нескольких десятков мегабайт	30–60 тактов работы
Оперативная память	Несколько гигабайт	50–100 тактов работы

Самая быстрая память – *регистровая*, но она небольшая по объему.

Кэш – это также небольшой¹ по объему, но скоростной модуль памяти, в который помещаются данные, часто используемые процессором. Использование кэш-памяти заметно ускоряет выполнение программы. Этому способствует *временная локальность* программ, означающая, что если произошло обращение к некоторой области памяти, то, скорее всего, в ближайшем будущем обращение к этому же участку памяти повторится. В результате при обращении к некоторой области памяти процессор сначала ищет данные в кэше. Если данные находятся (попадание в кэш), то они считываются из кэша, иначе (промах) данные считываются из оперативной памяти и в процессор, и в кэш. Для ускорения также используется свойство *пространственной локальности* программ: за обращением к не-

¹ Следует отметить, что объем кэш-памяти в современных компьютерах сравним с объемом оперативной памяти в компьютерах пятилетней давности.

которой странице памяти обычно следует обращение к соседним страницам. С учетом этого данные в кэш считываются блоками страниц.

По записи различают *сквозной кэш* – измененные данные помещаются в память немедленно – и *кэш с обратной записью*, в котором обновление оперативной памяти происходит по более сложным алгоритмам. В последнем случае может наблюдаться временное несоответствие содержимого кэша и оперативной памяти.

В современных процессорах обычно имеется до трех уровней кэша. *Кэш первого уровня* является неотъемлемой частью процессора, расположен с ним на одном кристалле и имеет минимальное время доступа. В современных многоядерных системах этот вид памяти принадлежит непосредственно ядру, а не всему процессору. *Кэш второго уровня* чаще всего также находится на кристалле процессора и принадлежит ядру. *Кэш третьего уровня* (если он есть), как правило, расположен отдельно, разделяется ядрами (процессорами) и может быть достаточно большим по объему (несколько мегабайт на ядро).

Уровни кэш-памяти отличаются и организацией. Кэш-память первого уровня *отображается непосредственно* и содержит отдельные области для хранения инструкций и данных. Кэш-память второго и третьего уровней обычно унифицирована, т. е. не делает различий между данными и инструкциями. Поскольку последовательный перебор всех строк кэша в поисках необходимых данных существенно увеличил бы время доступа, то память этих уровней чаще является *множественно-ассоциативной*. Для сокращения времени поиска ячейки оперативной памяти (ОП) жёстко привязываются к строкам кэш-памяти, т. е. в каждой строке могут быть данные из фиксированного набора адресов, при этом каждая ячейка ОП может быть ассоциирована с несколькими строками кэша.

Кэширование данных существенно увеличивает производительность современных компьютеров. Однако в многоядерных системах возникают проблемы согласования данных в кэшах ядер.

3. *Параллельная обработка данных* чаще всего реализует простую идею: если некое устройство выполняет одну операцию за единицу времени, то K операций оно выполнит за K единиц, а система из N устройств ту же работу выполнит за K/N единиц времени. Ярким примером операций, допускающих параллельную обработку, являются арифметические операции над векторами. В конце прошлого века процессоры, в набор команд которых входили операции над векторами, были широко распространены (например, компьютеры семейства Cray). В большинстве современных микропроцессоров имеются векторные расширения, в видеокартах используется параллельная обработка изображения.

С идеями параллельной обработки данных тесно связано и простое *дублирование устройств ВС*. Получило распространение использование нескольких однотипных обрабатывающих процессоров, а также многоядерных архитектур. Например, суперкомпьютер Roadrunner построен по гибридной схеме и содержит 6 120 двухъядерных процессоров AMD Opteron для вычислений и 12 240 процессоров IBM Cell 8i в специальных блэйд-модулях TriBlades. Кроме того, еще 442 двухъядерных процессора AMD Opteron задействованы для системных функций. В свою очередь, сам процессор IBM Cell 8i включает в себя одно универсальное ядро Power и восемь специальных ядер для операций с плавающей точкой.

Одной из реализаций параллельной обработки данных в архитектуре ВС можно считать *суперскалярность* многих современных процессоров (в том числе x86 начиная с Pentium, MIPS и т. д.). Суперскалярный процессор может выполнять несколько команд за один такт, причем распараллеливание потока команд происходит динамически на аппаратном уровне.

Развитие идей суперскалярности представлено *VLIW-архитектурой* (Very Long Instruction Word) процессоров. В этом случае одна инструкция процессора содержит набор операций, которые должны выполняться параллельно различными функциональными устройствами. Распараллеливание кода в таких архитектурах возложено на программиста или компилятор. В первом случае архитектуродоемкость процесса предполагает высокий уровень квалификации, а во втором – требуется перекомпиляция существующих кодов, практически исключается возможность влияния программиста на повышение производительности, падение которой неизбежно при универсальности, и не исключается проявление возможных ошибок компилятора. VLIW-архитектура реализована в микропроцессорах Крузо (Transmeta Crusoe), видеопроцессорах AMD, микропроцессорах архитектуры Intel Itanium и др.

4. *Конвейерная реализация обрабатывающих устройств* основана на том, что сложную операцию (даже некоторые команды процессора) можно разбить на несколько простых этапов¹. На каждом этапе функциональное устройство (ФУ), выполнив свою работу, передает результат следующему, одновременно принимая от предыдущего новую порцию входных данных. Организовывая цепочку из таких ФУ, получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций.

Предположим, что в операции над некоторым набором операндов можно выделить K микроопераций, каждая из которых выполняется за одну

¹ Например, для сложения двух вещественных чисел, представленных в форме с плавающей запятой, необходимо выполнить множество мелких операций: сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т. п.

единицу времени. Если есть одно неделимое последовательное универсальное устройство, то M наборов операндов оно обработает за $K \times M$ единиц. Если каждую микрооперацию выделить в отдельный этап (ступень) конвейерного устройства, то на K -й единице времени на разной стадии обработки будут находиться первые K наборов операндов. Все M наборов операндов будут обработаны за $K + (M - 1)$ единиц времени – ускорение по сравнению с последовательным устройством почти в K раз (по числу ступеней конвейера).

Проиллюстрируем преимущество применения конвейера на примере простого пятиуровневого¹ конвейера команд в RISC-процессорах (рис. 1.3).

Instruction 1					Instruction 2					Instruction 3					Instruction 4					Instruction 5									
IF	ID	EX	MEM	WB	IF	ID	EX	MEM	WB	IF	ID	EX	MEM	WB	IF	ID	EX	MEM	WB	IF	ID	EX	MEM	WB					
					<i>IF</i>					<i>ID</i>					<i>EX</i>					<i>MEM</i>					<i>WB</i>				
<i>TICK 1</i>					IF 1					N/A					N/A					N/A					N/A				
<i>TICK 2</i>					IF 2					ID 1					N/A					N/A					N/A				
<i>TICK 3</i>					IF 3					ID 2					EX 1					N/A					N/A				
<i>TICK 4</i>					IF 4					ID 3					EX 2					MEM 1					N/A				
<i>TICK 5</i>					IF 5					ID 4					EX 3					MEM 2					WB 1				
<i>TICK 6</i>					BUBBLE					ID 5					EX 4					MEM 3					WB 2				
<i>TICK 7</i>					BUBBLE					BUBBLE					EX 5					MEM 4					WB 3				
<i>TICK 8</i>					IF 6					BUBBLE					BUBBLE					MEM 5					WB 4				
<i>TICK 9</i>					IF 7					ID 6					BUBBLE					BUBBLE					WB 5				

Рис. 1.3. Исполнение 5 команд в RISC-процессорах: последовательное (вверху) и с помощью конвейера команд (внизу). Показано продвижение по конвейеру «пузыря»

В этом случае принято, что выполнение типичной инструкции разделено на пять этапов (ступеней конвейера):

IF (Instruction Fetch) – по текущей позиции счетчика команд выбор очередной команды из памяти и размещение ее в регистре команд, увеличение счетчика команд;

ID (Instruction Decoding) – декодирование команды, определение ее типа и типов операндов, при необходимости параллельная загрузка операндов из регистрового файла в регистры;

EX (Execute) – выполнение операции над соответствующими регистрами (для команд типа Register-Register ALU instruction и Regis-

¹ Количество этапов, на которые разбивается выполнение процессорной команды, сильно варьируется; например, в разных моделях процессоров x86 конвейер команд имеет от 2 ступеней для i8088 до нескольких десятков у Pentium.

ter_Immediate ALU instruction) или вычисление эффективного адреса для команд загрузки или записи (Load / Store instructions);

MEM (Memory access) – непосредственная работа с памятью для команд загрузки / записи;

WB (Write Back cycle) – запись результата команды в регистровый файл (из памяти в случае операции загрузки или непосредственно из ALU).

Если считать, что выполнение каждой ступени конвейера занимает один такт, то конвейерное выполнение пяти команд займет всего 9 тактов (рис. 1.3), а их последовательное выполнение заняло бы 25 тактов. Отметим, что для успешного функционирования конвейера необходимо, чтобы данные для первой операции поступали непрерывно: иначе неизбежно возникновение «пузырей» (bubble), снижающих эффективность процесса.

Конечно, конвейерную обработку можно заменить обычным параллелизмом, для чего надо продублировать универсальное устройство по числу ступеней конвейера. Каждое такое устройство должно: 1) быть универсальным, т. е. уметь делать операции всех ступеней конвейера; 2) сделать все операции за то же время, которое работает конвейер. Это многократно увеличивает объем, энергопотребление и стоимость оборудования.

5. *Использование специализированных устройств* широко распространено в современных ВС. Например, блэйд-модули уже упоминавшегося суперкомпьютера Roadrunner содержат процессоры IBM Cell 8i, включающие в себя наряду с универсальным ядром Power восемь специальных ядер для операций с плавающей точкой.

Современные видеокарты содержат графические процессоры (GPU), которые по сложности не уступают центральному процессору компьютера, но имеют архитектуру, максимально нацеленную на увеличение скорости расчета текстур и сложных графических объектов. В настоящее время GPU – это массивно-параллельное программируемое вычислительное устройство с высоким быстродействием, большим объемом собственной оперативной памяти. GPU может использоваться как сопроцессор, пригодный для решения широкого круга задач, мало связанных с графикой. В связи с этим появился термин GPGPU (General Purpose Graphics Processor Unit) – графические процессоры общего назначения.

Более того, наряду с видеокартами получают распространение «ускорители физики» – выделенный специализированный процессор, предназначенный для обработки динамичного видеоизображения, связанного с динамикой жидкости, твердых и мягких тел, обнаружением столкновений, симуляцией волос, меха и ткани, разломов объектов.

1.2. Основные классы современных параллельных компьютеров

Современные суперкомпьютеры поражают воображение. В первом параграфе мы уже упоминали характеристики высокопроизводительной вычислительной системы Roadrunner, занимавшей первое место в 32-м списке¹ (вышедшем в ноябре 2008 года) самых мощных компьютеров мира.

Другим популярным решением является семейство суперкомпьютеров BlueGene. Например, суперкомпьютер BlueGene/L (eServer Blue Gene Solution, рис. 1.4–1.6) от IBM Ливерморской национальной (радиационной) лаборатории Лоренса (США) в течение двух лет (2006–2007) занимал первую строчку² в top500. Суперкомпьютер объединяет 106 496 двухпроцессорных узла (всего 212 992 процессора) с общей оперативной памятью в 53 TB (терабайта) и пиковой производительностью 596,4 TFlops (5,964e+14 Flops). На тесте LINPACK³ BlueGene/L показал производительность 478,2 TFlops. Узлы BlueGene/L сконфигурированы в 32×32×64-трехмерный тор: каждый узел непосредственно связан с шестью соседними узлами. Древовидная структура поддерживает быстрые (несколько микросекунд) операции свертки данных (суммирование всех элементов массива, нахождение максимального и т.д.) на всех 212 992 процессорах. Обмен с дисковой системой происходит со скоростью 1 024 гигабит в секунду.

Ясно, что создание и обслуживание таких вычислительных монстров – это не простое любопытство небольших коллективов ученых-энтузиастов, а отвечающая современным экономическим запросам технология.

Автомобилестроение, нефте- и газодобыча, фармакология, прогноз погоды и моделирование изменения климата, сейсморазведка, проектирование электронных устройств, синтез новых материалов, крупнейшие мировые системы бронирования – вот неполный список областей челове-

¹ На сайте top500.org можно найти выпускаемый два раза в год (в июне и ноябре) рейтинг суперкомпьютеров. Рейтинг строится на основе сравнений пиковой производительности и производительности, показанной по тесту LINPACK. Последний раз Roadrunner упомянут в 40-й редакции списка от ноября 2012 года на 22-м месте.

² Развитие высокопроизводительной техники стремительно. Последний раз BlueGene/L упоминался в 36-й редакции списка top500 от ноября 2010 года на 12-м месте. Однако архитектурное решение BlueGene оказалось очень удачным, а суперкомпьютеры подобной архитектуры регулярно занимают высокие строчки в top500.

³ Тест решает случайно заданную систему линейных уравнений с плотной матрицей методом LU-разложения, используя 64-битную арифметику с плавающей точкой, что позволяет судить о реальной производительности вычислительной системы. Существует много публикаций, критикующих такой подход к оценке реальной производительности суперкомпьютеров, но общепринятой альтернативы тесту до сих пор не предложено.

ческой деятельности, в которых использование суперкомпьютеров действительно необходимо.

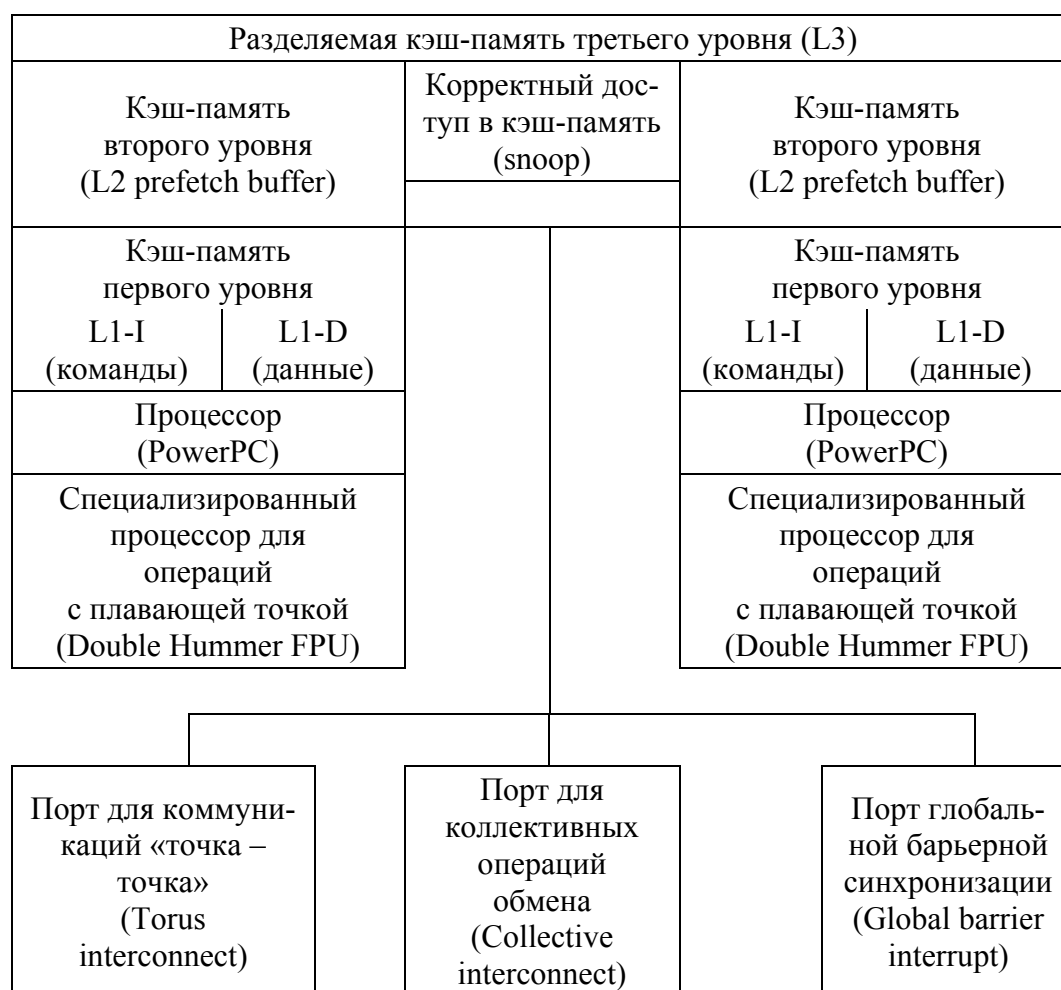


Рис. 1.4. Устройство одного узла в архитектуре BlueGene/L

Самой известной схемой классификации компьютерных архитектур является *таксономия Флинна* [60–61], предложенная еще в 1966 году. В ее основу положены *способы взаимодействия компьютера с потоками команд и данных*. Согласно Флинну имеется два класса параллельных архитектур, которые впоследствии были дополнены еще одной (систолические массивы), в эту же классификацию включена архитектура обычного однопроцессорного компьютера. Таким образом, в настоящее время говорят о следующей таксономии Флинна.

1. SISD (Single Instruction Stream – Single Data Stream) – это обычные последовательные компьютеры, когда в каждый момент времени выполняется только одна операция над одним элементом данных. Возможно включение в состав ВС нескольких процессоров, но каждый работает со своим набором данных, относящихся к своей независимой программе.

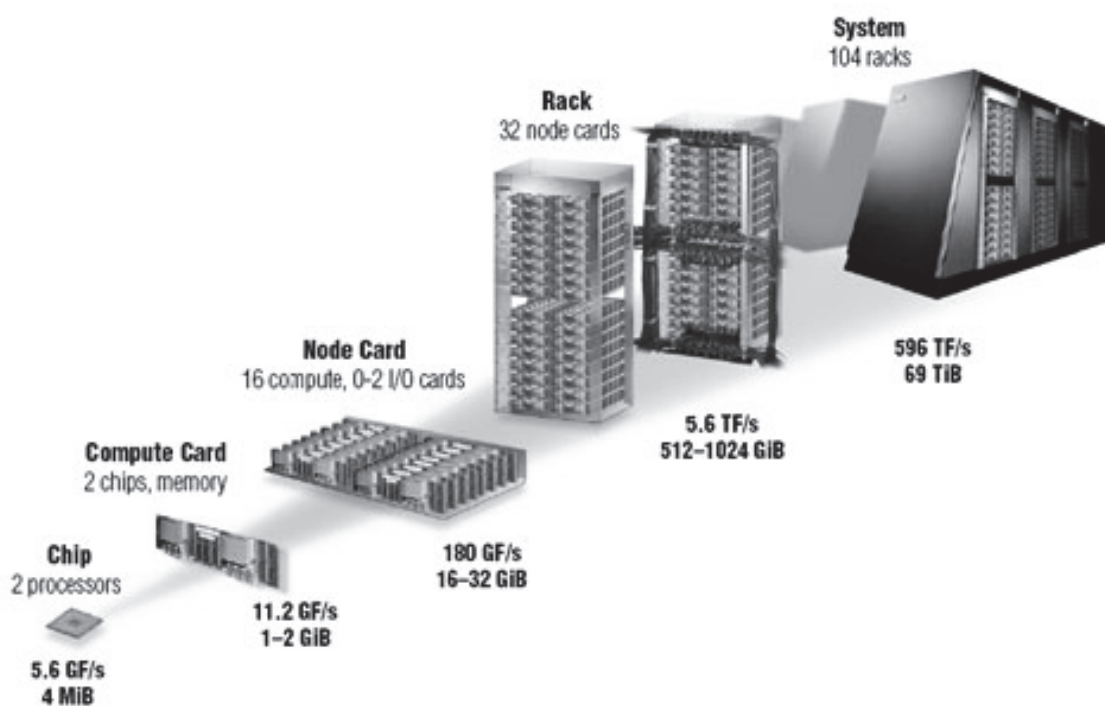


Рис. 1.5. Архитектура суперкомпьютера BlueGene/L
(https://asc.llnl.gov/computing_resources/bluegenel/configuration.html)

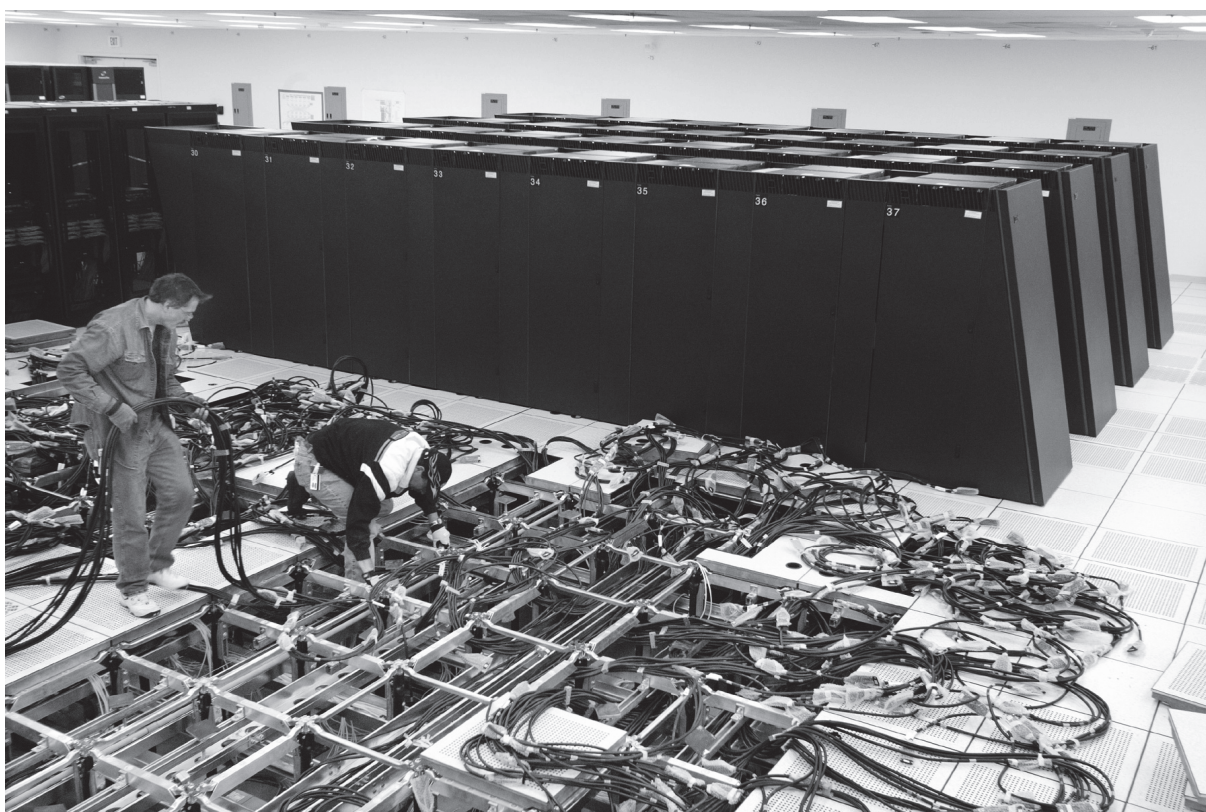


Рис. 1.6. Монтаж суперкомпьютера BlueGene/L
(https://asc.llnl.gov/computing_resources/bluegenel/images/cables.jpg)

2. SIMD (Single Instruction Stream – Multiple Data Stream) состоят из одного командного процессора (управляющего модуля, контроллера) и нескольких модулей обработки данных (процессорных элементов, ПЭ). Управляющий модуль принимает, анализирует и выполняет команды. Если в команде встречаются данные, то контроллер рассылает команду на все ПЭ, и команда выполняется над данными каждого процессорного элемента. Яркими представителями этого класса являются векторные компьютеры, а также матричные процессоры. Часто такие архитектуры специализированы под решение конкретных задач, допускающих матричное представление, например, задач обработки изображений.

3. MISD (Multiple Instruction Stream – Single Data Stream). Систолический массив процессоров, в котором процессоры находятся в узлах регулярной решетки. Роль ребер играют межпроцессорные соединения, все ПЭ управляются общим тактовым генератором. В каждом цикле любой ПЭ получает данные от своих соседей, выполняет одну команду и передает результат соседям.

4. MIMD (Multiple Instruction Stream – Multiple Data Stream). Этот класс наиболее богат примерами успешных реализаций. Это кластеры рабочих станций, симметричные параллельные ВС и пр.

Классификация Флинна не дает исчерпывающего описания разнообразных MIMD-архитектур. Например, системы с разделяемой памятью относятся как к MIMD-, так и SIMD-архитектуре. Поэтому рассматриваются и другие способы классификации параллельных компьютеров, например, Хокни, Фенга, Хендлера, Шнайдера, Скилликорна [9].

Самой простой классификацией суперкомпьютеров является деление их *по способу взаимодействия процессоров с оперативной памятью* (рис. 1.7).

1. Векторно-конвейерные компьютеры. Конвейерные функциональные устройства и набор векторных команд – это две особенности таких ВС. В отличие от традиционного подхода векторные команды оперируют целыми массивами независимых данных, что позволяет эффективно загружать доступные конвейеры.

Векторно-конвейерная архитектура процессора (PVP, Pipe Vector Processor) может использоваться как в компьютерах с одним процессором, так и в многопроцессорной системе.

2. Параллельные компьютеры с общей (разделяемой) памятью. Вся оперативная память таких компьютеров разделяется несколькими одинаковыми процессорами. В данное направление входят многие современные многопроцессорные и многоядерные SMP-компьютеры (Single Memory Processor) или, например, отдельные узлы компьютеров HP Exemplar и Sun StarFire.

В мультипроцессоре с общей (разделяемой) памятью (SMP) процессоры и оперативная память связаны с помощью соединительной сети (рис. 1.8). Процессоры совместно используют память, но каждый из них имеет собственный кэш.

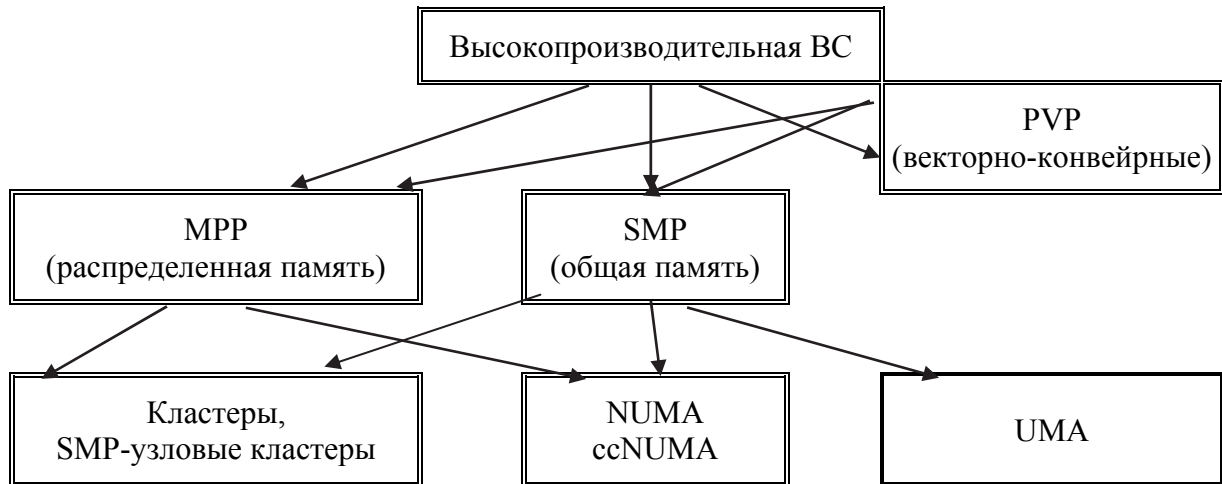


Рис. 1.7. Классификация суперЭВМ



Рис. 1.8. Структура мультипроцессора с разделяемой памятью

Если число процессоров относительно невелико (от двух до тридцати), то связующая сеть может быть реализована в виде шины памяти или матричного коммутатора. Поскольку время доступа каждого из процессоров к любому участку памяти одинаково, то такой мультипроцессор называется *однородным (UMA, uniform memory access)*.

В мультипроцессорах с разделяемой памятью, включающих десятки и сотни процессоров, память организована иерархически. Связующая сеть имеет вид древообразного набора переключателей и областей памяти (межкластерные шины, Butterfly и пр.). Поскольку одна часть памяти ближе к определенному процессору, другая дальше от него, то возникает неоднородность времени доступа, зато снижается нагрузка на каждую шину или

коммутатор. Такая архитектура называется *неоднородной (NUMA, non uniform memory access)*.

В SMP-компьютерах каждый процессор имеет собственную кэш-память (кэш), что порождает *проблему согласованности кэша*. Если два процессора обращаются к одной области памяти примерно одновременно и один из них записывает данные, то кэш-память этих процессоров будет содержать разные данные. Необходимо либо обновить кэш, либо признать его содержимое недействительным. Протоколы согласования реализуются аппаратно. NUMA-архитектура, обеспечивающая аппаратную поддержку согласованности кэш-памяти, называется *ccNUMA – cash caherent non uniform memory access*.

В SMP-архитектурах также возникает *проблема согласованности оперативной памяти*, решение которой опирается на одну из следующих моделей. *Последовательная согласованность* гарантирует, что обновление памяти происходит в некоторой последовательности, причем каждому процессору «видна» одна и та же последовательность. Это самая сильная модель. *Согласованность процессоров* обеспечивает упорядоченность записи в память, выполняемой каждым процессором, но порядок записей, инициированных разными процессорами, для других процессоров может выглядеть по-разному. Самая слабая модель – *согласованное освобождение*, при которой оперативная память обновляется в указанных программистом точках синхронизации.

В последнее время быстрыми темпами развиваются многоядерные архитектуры, подразумевающие наличие не только общей оперативной памяти, но и общего кэша второго или третьего уровня.

3. Массивно-параллельные компьютеры с распределенной памятью. Компьютеры этого класса обычно состоят из серийных микропроцессоров (каждый со своей локальной памятью), соединенных посредством коммуникационной среды.

Достоинства такой архитектуры очевидны – легкая масштабируемость, возможность простого расчета оптимальной конфигурации и т. п. Однако межпроцессорное взаимодействие в компьютерах этого класса идет намного медленнее, чем происходит локальная обработка данных самими процессорами.

В *мультипроцессорах с распределенной памятью* (MPP, Massively Parallel Processor) каждый процессор имеет свою оперативную память. В такой архитектуре процессоры взаимодействуют через соединительную сеть (рис. 1.9), принимая и передавая сообщения, и, следовательно, отсутствуют проблемы согласованности кэша и памяти. Соединительная сеть может быть организована в гиперкуб или матричную структуру и является высокоскоростным путем связи между процессорами.

Наибольшее распространение в рамках МРР-архитектуры получили высокопроизводительные кластерные системы – все лидеры top500 являются представителями именно такой технологии.

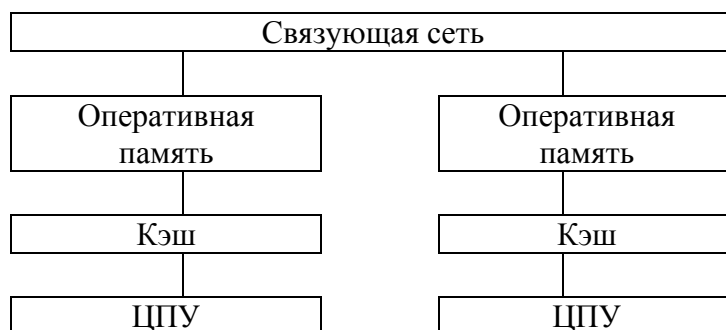


Рис. 1.9. Структура ЭВМ с распределенной памятью

Современный высокопроизводительный кластер обычно строится следующим образом. Процессорные элементы (вычислительные узлы) состоят из одного или нескольких, чаще всего многоядерных, процессоров с общей памятью. Узлы между собой соединены специальной коммуникационной сетью с малой латентностью и высокой пропускной способностью. Это может быть коммутируемая последовательная шина или кольцо, одно- или многомерные торы и пр. (наиболее распространены технологии InfiniBand, SCI, Mirennet, Gigabit Ethernet). Часто сеть строится на основе специальных микросхем, включающих модуль прямого доступа к памяти, процессор передачи данных и управления, высокоскоростной маршрутизатор, создающий несколько высокоскоростных соединений с соседями. Для связи с управляющим узлом (или узлами), управления и мониторинга системы вычислительные узлы объединяются вспомогательной сетью, которая может быть более медленной и дешевой. Узлы могут быть снабжены собственной дисковой памятью, возможно, с высокоскоростными подсистемами ввода-вывода, при этом дополнительно к системе добавляются серверы доступа и файловый сервер с дисковыми массивами.

В настоящее время получили распространение термины, связанные с префиксом «блейд» (блейд-модули в суперкомпьютерах, блейд-сервера, блейд-системы и пр.). Термин «блейд» происходит от английского «лезвие» и прямо связан с семантикой – это не новая технология, а просто способ компактной организации суперкомпьютера¹ или стойки серверов. Достига-

¹ Принята специальная единица измерения высоты оборудования, размещаемого в стойке (шкафу): 1U (unit) равен 4,445 см, или 1,75 дюйма. В стандартной 19-дюймовой стойке можно разместить максимально 42 сервера, что может предполагать до 140 кабелей. Блейд-технология позволяет в настоящее время довести количество плат в стойке до 100 и уменьшить количество кабелей до трех.

ется экономия пространства в основном путем исключения из самой платы различных вспомогательных (не относящихся непосредственно к вычислениям) устройств: питание, охлаждение, сетевые подключения, подключения жёстких дисков, межсерверные соединения и управление выносятся за пределы платы и обслуживают стойку серверов.

4. Гибридные архитектуры. Наиболее общая комбинация – машина с поддержкой *распределенной разделяемой памяти*, т. е. распределенной реализации абстракции разделяемой памяти.

В конце параграфа кратко обсудим высокопроизводительные технологии, представленные в мире, которые к архитектуре суперкомпьютера отнести нельзя. Одним из представителей систем с распределенной памятью является *кластер рабочих станций*, объединенных коммуникационной сетью, желательно, но вовсе не обязательно, высокоскоростной. Такие кластеры обычно рассматривают как относительно дешевую альтернативу крайне дорогим суперкомпьютерам. Проект Beowulf¹ предполагал технологию построения кластера из широко распространённого аппаратного обеспечения, работающего под управлением свободно распространяемой операционной системы.

Глобализация кластерной технологии приводит к понятию *метакомпьютинга*. Интернет очень напоминает кластерную архитектуру: множество узлов с собственными процессорами, оперативной и внешней памятью, устройствами ввода-вывода, соединенные друг с другом коммутационным оборудованием и линиями передачи данных. Энтузиастами метакомпьютинга решаются задачи по поиску все новых простых чисел, повышению точности значения числа π , обнаружению сигналов внеземных цивилизаций и даже поиску формулы лекарства от СПИДа.

В настоящее время в научном сообществе распространена *технология грид*² – согласованная, открытая и стандартизованная компьютерная среда, которая обеспечивает гибкое, безопасное, скоординированное разделение вычислительных ресурсов и ресурсов хранения информации в рамках некоторой сети. В рамках грид-проектов предполагается объединять и распределять вычислительные ресурсы не энтузиастов-одиночек, а суперкомпьютерных центров. Получив закрытый ключ и сертификат для вхождения в грид-инфраструктуру, пользователь может запускать со своего рабочего компьютера трудоемкую задачу, не имея представления о том, в каких вычислительных центрах она решается и где хранятся полу-

¹ Название происходит от одного из Linux-кластеров в научно-космическом центре NASA. Начало проекта – 1994 год.

² Идеи грид-системы были собраны и объединены Иэном Фостером, Карлом Кессельманом и Стивом Тики. В настоящее время Карл Кессельман является директором Центра грид-технологий Института информатики Университета Южной Калифорнии.

ченные данные. Грид-системы гарантируют безопасность и защиту информации. Ярким примером является научно-техническая программа «СКИФ-ГРИД», выполняемая совместно Россией и Белоруссией. Цель программы – «освоение и адаптация передовых наукоемких технологий на перспективных суперкомпьютерных платформах, оптимизация суперкомпьютерных конфигураций семейства “СКИФ”, ориентированных на построение на их основе Грид-компьютерных сетей...»¹.

Технология *облачных вычислений* – это еще один современный подход к обеспечению сетевого доступа по требованию к общему пулу конфигурируемых вычислительных ресурсов. В отличие от грид-технологий, связанных, прежде всего, с научными интересами, облачные сервисы обслуживают широкий круг потребителей вычислительных услуг от коммерческих или государственных организаций до индивидуальных пользователей.

1.3. Разработка параллельных приложений

Существуют три широко используемых и частично перекрывающихся подхода к организации параллельных приложений, определяющих три соответствующих им класса программ (рис. 1.10).

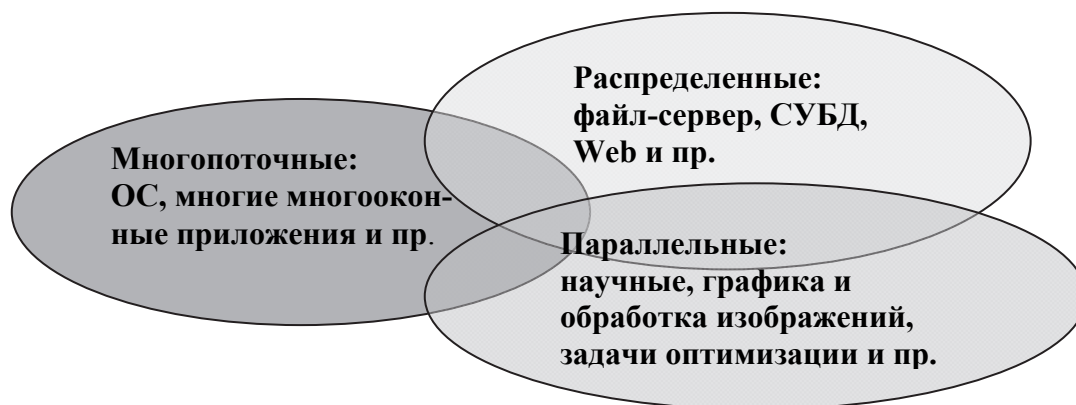


Рис. 1.10. Классы приложений

1. Многопоточные приложения. Термин *многопоточный* обычно означает, что программа содержит больше процессов/потоков², чем суще-

¹ URL: <http://skif-grid.botik.ru/>.

² Мы оставим за рамками обсуждения различия между терминами «процесс» и «поток». Здесь и далее в этой главе эти термины считаются взаимозаменяемыми. Затем под *процессом* понимается выполняемая в системе программа, решающая некоторую задачу, владеющая и/или претендующая на некоторые ресурсы системы. Ресурсы процессу могут выделяться как в безраздельное пользование (например, адресное пространство), так и разделяться с другими процессами (например, процессор). *Поток* –

ствуется процессоров для их выполнения. В результате некоторые процессы приложения выполняются по очереди.

Примерами многопоточных программ являются операционные системы (ОС), системы разделения времени, системы реального времени, многие оконные приложения. Эти системы разрабатываются многопоточными, поскольку составляющие их задачи могут планироваться и выполняться независимо. Например, в MS Word один поток отвечает за проверку орфографии, другой – за пересчет ссылок и полей, третий – за ввод данных и т. д.

Основной проблемой реализации таких программ является борьба за разделяемые ресурсы, например, процессор, память, устройства ввода-вывода и другие.

2. Распределенные системы. В таких системах компоненты программ выполняются на разных вычислительных машинах, связанных сетью, возможно, работающих под разными ОС. Основным способом взаимодействия процессов в распределенных системах является обмен сообщениями. Спецификой процессов является их относительная независимость и возможность обслуживания одним процессом различных клиентов. Подобное обслуживание обычно происходит по поступающим запросам.

Примерами распределенных систем могут служить файловые серверы в сети, СУБД, веб-серверы в сети Интернет, отказоустойчивые системы, которые продолжают работать независимо от сбоев в компонентах.

Распределенные системы необходимы для организации доступа к территориально удаленным данным, интеграции и управления данными, распределенными по своей сути, повышения надежности. Как правило, такие системы основаны на технологии клиент-сервер, а их компоненты сами являются многопоточными программами.

Основной проблемой реализации распределенных программ является обмен информацией о состоянии разделяемого ресурса.

3. Параллельные (согласованные) вычисления. Цель таких приложений – совместно и быстро решить данную задачу, причем допустимы как многопоточный, так и распределенный подходы.

Примерами согласованных параллельных вычислений являются различные научные вычисления (например, численные модели математической физики), графика и обработка изображений (включая создание спецэффектов в кино), крупные комбинаторные и оптимизационные задачи, интеллектуальный анализ данных.

это «облегченный» процесс, т. е. потоки имеют собственные счетчики команд, стеки выполнения, но работают в общем адресном пространстве.

Параллельные программы требуют больших вычислительных мощностей. Обычно число *процессов в параллельных программах равно числу процессоров (ядер)*.

В таксономии Флинна уже прослеживается два основных подхода к построению параллельных приложений.

1. Параллелизм по данным основан на выполнении процессами (потоками) одних и тех же действий над собственной частью данных. В этом случае программист задает опции векторной или параллельной оптимизации, директивы параллельной компиляции, а собственно векторизацию или распараллеливание выполняет транслятор. Существуют также специализированные языки параллельных вычислений. Параллелизм по данным реализует SIMD-технология. Для такого подхода характерно, что обработкой данных управляет одна программа, но процессы слабо синхронизированы (каждый процесс выполняет один код, но нет гарантии, что в заданный момент времени все процессы выполняют одну и ту же инструкцию).

2. Параллелизм по задачам предусматривает разбиение вычислительной задачи на несколько самостоятельных подзадач. Компьютер при этом представляет MIMD-машину. Для каждой подзадачи, как правило, пишется своя программа. Подзадачи в ходе работы обмениваются данными, а также согласуют свою работу. При программировании контролируется распределение данных между процессами/потоками и вычислительной нагрузки между процессорами, что повышает трудоемкость разработки и отладки программы.

Как правило, выбор метода распараллеливания основан на свойствах самой задачи или выбранного алгоритма ее решения. Так, для сеточных вычислений характерен параллелизм по данным, однако можно выбрать такой алгоритм решения (например, метод прогонки для решения СЛАУ с трехдиагональной матрицей), который испортит весь параллелизм исходной задачи. Задача обработки изображения может обладать внутренним параллелизмом по задачам, а каждая задача – параллелизмом по данным.

Процесс проектирования параллельных приложений обязательно включает следующие три составляющие: *декомпозиция – связь – синхронизация* (ДСС).

Декомпозиция – это процесс разбиения прикладной задачи на части. Иногда декомпозиция естественным образом следует из природы самой задачи, иногда она определяется разработчиком.

Можно проводить разбиение *на основе логики действий* (например, сортировка, поиск, ввод-вывод, вычисление), *на основе логики ресурсов* (например, работа с файлом, принтером, базой данных), *на основе логики данных* (например, обработка массива по строкам, столбцам или блокам).

Чем меньше размер подзадач, полученных на этом этапе, и чем больше их количество, тем более гибким получается параллельный алгоритм и тем легче обеспечить равномерную загрузку процессоров. При необходимости всегда можно провести обратную операцию – *укрупнение*.

Параллелизм можно организовать на *уровне команд* (рис. 1.11). Этот вид параллелизма обычно обеспечивается компилятором, ОС и внутренней архитектурой процессора. Обсуждение этих вопросов выходит за рамки нашего пособия.

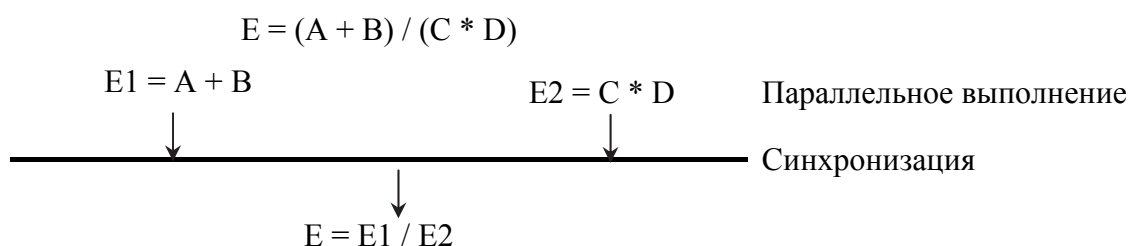


Рис. 1.11. Синхронизация на уровне команд

Следующий уровень параллелизма в итеративных программах – *цикл*. Не всякий цикл можно выполнять параллельно [9]. Решение об этом принимает, как правило, программист, например, с помощью директив компиляции.

Дальнейшее использование параллелизма проводится на *уровне подпрограмм* (функций). Решение подзадач можно оформить отдельными функциями, выполнение которых распределяется по потокам. При наличии нескольких процессоров выполнение ряда потоков может протекать параллельно. Этот уровень требует тщательного проектирования синхронизации и взаимодействия, которые, в свою очередь, во многом определяются архитектурой вычислительной системы.

Как параллелизм этого же уровня можно представить и некоторую разновидность асинхронного выполнения SIMD-программ, при котором один и тот же код программы запускается на разных вычислительных узлах среды с распределенной памятью, поддерживающей обмен сообщениями.

В данном учебном пособии в основном обсуждается создание приложений, поддерживающих параллелизм уровня циклов и подпрограмм.

Дальнейшее укрупнение уровня параллелизма – *объекты и приложения* – характерно для распределенных приложений, которые в нашем учебном пособии не рассматриваются.

Планирование связей, или коммуникаций, между частями приложения – следующий этап проектирования. Методы связывания, как правило, определяются архитектурой вычислительной системы (разделяемые переменные, каналы, передача сообщений и т. д.), а их конкретная программ-

ная реализация диктуется выбранным инструментом программирования – языком или библиотекой.

Поскольку множество потоков программы работают в рамках одной задачи, то их функционирование необходимо координировать, поэтому важным моментом проектирования является *синхронизация*. Следует предусмотреть, какие задачи могут выполняться параллельно, а какие – последовательно и в каком порядке; если некоторые части программы закончили работу раньше, то следует ли им заняться выполнением других заданий. Необходимо учесть и то, как разные подзадачи «узнают», что решение достигнуто.

При некорректной организации связей и синхронизации обычно появляются следующие проблемы.

«Гонка данных» возникает из-за того, что несколько процессов одновременно пытаются обновить и использовать общие данные. Проблема наиболее очевидна, если данные одновременно перезаписываются. Однако «гонка данных» часто возникает и в случае, когда один процесс неоднократно использует данные, обновляемые другим процессом. При отсутствии согласованности задача-потребитель может использовать устаревшие данные (при опережающем чтении) или утерять данные (производитель может при записи затереть еще не использованные данные).

Некорректное планирование последовательности выполнения задач приводит и к другой проблеме – «живой блокировке», при которой задача может никогда не получить возможность выполниться. Также эту проблему иногда называют «ситуацией бесконечной отсрочки».

«Мертвая (взаимная) блокировка» – еще одна ловушка, связанная с ожиданием. В простейшем случае взаимная блокировка возникает, если «задача 1» ждет данных (сигнала) от «задачи 2», которая, в свою очередь, ожидает данных (сигнала) от «задачи 1». В общем случае ситуация может оказаться еще более запутанной.

Все эти проблемы должны быть выявлены и разрешены на стадии проектирования параллельного приложения.

1.4. Программные средства

Эффективность разработки параллельных программ во многом зависит от наличия соответствующего программного инструментария. Разработчик должен эффективно использовать все методы от средств анализа и выявления параллелизма до трансляторов, отладчиков и ОС, обеспечивающих надежную работу программы на многопроцессорных вычислительных системах.

Ниже дан краткий обзор языков параллельного программирования, библиотек и систем разработки параллельных программ [16, 28, 43–44, 48, 64].

Библиотека PTHREAD разработана в 90-х годах прошлого века под эгидой организации POSIX (Portable Operating System Interface – интерфейс переносимой ОС). Она определяет стандартный набор функций языка Си для многопоточного программирования. В распоряжении программиста оказывается большой набор функций создания и управления потоками и набором их атрибутов, а также различные методы синхронизации потоков (блокировки, мьютексы, семафоры, барьеры).

Язык программирования Java объединяет возможности интерпретируемых, объектно ориентированных и параллельных языков, добавляя несколько новых. Он поддерживает как многопоточное, так и распределенное программирование.

Одним из наиболее популярных средств программирования многопроцессорных компьютеров с общей памятью в настоящее время является *технология OpenMP*. При ее использовании за основу берется последовательная программа. Для создания ее параллельной версии применяется набор директив, процедур и переменных окружения. Технология OpenMP представляет программу как цепочку параллельных и последовательных областей, используя схему FORK/JOINT для поддержки параллелизма. Все порожденные нити исполняют один и тот же код параллельной области. Переменные программы разделяются на *общие* (SHARED) и *локальные* (PRIVATE). Стандарт OpenMP разработан для языков Fortran, Си, C++. Реализация стандарта доступна как для UNIX-платформ, так и в среде Windows.

Язык программирования CSP (Communicating Sequential Processes – взаимодействующие последовательные процессы) впервые был описан в 1978 году Тони Хоаром. Его первое описание содержало идеи синхронной передачи сообщений и защищенного взаимодействия. Язык CSP повлиял на разработку таких языков программирования, как Occam и Ada. Его последняя версия стала формальным языком для моделирования и анализа поведения параллельных систем.

Язык программирования Occam начал разрабатываться в середине 1980-х годов для транспьютеров, являясь, по сути, их машинным языком. Язык содержит очень малое число механизмов, что и определило его название (от выражения «бритва Оккама»). Базовыми элементами языка являются декларации и три примитивных процесса (присваивание, ввод и вывод), которые объединяются в обычные процессы с помощью трех типов конструкторов (последовательный, параллельный и защищенного взаимодействия).

Система программирования Linda разработана в середине 1980-х годов в Йельском университете США. В рамках системы параллельная программа – это множество параллельных процессов, каждый из которых работает как последовательная программа. Все процессы имеют доступ к общей памяти, единицей которой является *кортеж* – упорядоченная последовательность значений. Процессы в системе Linda никогда не взаимодействуют друг с другом явно, их общение идет всегда опосредованно, через пространство кортежей, которое можно считать виртуальной ассоциативной памятью. По замыслу создателей Linda, для того чтобы любой последовательный язык стал средством параллельного программирования, достаточно добавить в него только четыре функции: извлечь/прочитать элемент из пространства кортежей, поместить элемент в пространство кортежей, вычисляя его последовательно или параллельно. Несколько языков программирования, включая Си и FORTRAN, были дополнены примитивами Linda.

При использовании *библиотеки MPI* (Message Passing Interface – интерфейс передачи сообщений) процессы распределенной программы записываются на таком последовательном языке, как Си или FORTRAN. Их взаимодействие или синхронизация задаются с помощью вызова процедур библиотеки. Каждый процессор выполняет копию одной и той же программы, использующей функции MPI. Каждый экземпляр программы может определить собственный идентификатор и, следовательно, выполнять действия, отличные от других.

Система разработки и выполнения параллельных программ PVM (Parallel Virtual Machine – параллельная виртуальная машина) разработана в США в 90-х годах прошлого века. Она позволяет объединить разнородный набор компьютеров, связанных сетью в общий вычислительный ресурс. Единицей параллелизма в PVM является задача, состояние которой изменяется (вычисления, обмен данными и пр.). Выполнение PVM-программы порождает несколько процессов, которые взаимодействуют посредством обмена сообщениями. Система поддерживает языки Си, FORTRAN, Perl, Java и даже математический пакет Matlab.

HPF (High Performance FORTRAN – высокопроизводительный FORTRAN) – это язык, параллельный по данным. Он является расширением языка FORTRAN-90 для разработки параллельных программ. Основные компоненты HPF – параллельное по данным присваивание массивов, директивы компилятора для управления распределением данных и операторы записи и синхронизации параллельных циклов.

Система разработки и выполнения параллельных программ DVM разработана в ИПМ им. М. В. Келдыша РАН как расширение языков Си и FORTRAN. Последовательная программа, написанная на одном из этих языков, дополняется директивами, оформленными как параметры макроса,

который в последовательной программе расширяется в пустую строку, что позволяет поддерживать одну программу для последовательной и параллельной версий. DVM-система реализует параллелизм по данным для компьютеров с распределенной, общей памятью, а также гибридной архитектуры, при которой каждый узел распределенной вычислительной системы является многоядерным (многопроцессорным), расширенным графическими процессорами. Директивы DVM-системы управляют распределением данных, вычислений и удаленным доступом к данным. Среда дополнена инструментами запуска, отладки и анализа производительности.

Языки, параллельные по данным (*C**, *ZPL*, *NESL*), поддерживают стиль программирования, в котором все процессы выполняют один и тот же код на разных частях данных. Они значительно упрощают обработку массивов и матриц. Несмотря на явное взаимодействие процессов, в таких языках синхронизация часто поддерживается неявно, после каждого оператора. Большинство языков, параллельных по данным, создавалось под определенную архитектуру ВС (например, *C** тесно связан с архитектурой CM-1, первой SIMD Connection Machine).

Объектно ориентированные системы, как правило, поддерживают многопоточность. Например, библиотека *C Qt* позволяет осуществлять управление потоками добавлением объектов класса *QThread*.

Особо следует отметить объектно ориентированные языки, например *C#*, *Distributed Java*, поддерживающие CORBA (Common Object Request Broker Architecture) – стандарт для определения отношений, взаимодействий и связей между распределенными объектами.

В языках упреждающих вычислений параллелизм реализован с помощью параллельного выполнения вычислений еще до того, как их результат потребуется для продолжения выполнения программы. Таковым, например, является *Multilisp* – параллельная версия известного декларативного языка *Lisp*.

Еще одним неимперативным языком для параллельного программирования является разработка компании Ericsson – функциональный язык *Erlang*. Изначально сотрудники лаборатории Ericsson Computer Science Laboratory реализовали возможности параллелизма логического языка Prolog. Перестав быть диалектом Prolog'а в 1990 году, разработка получила название¹ и собственный синтаксис. В настоящее время язык Erlang, построенный на идеологии передачи сообщений, считается языком многопоточного и распределенного программирования высокой надежности.

¹ По фамилии датского математика Агнера Краупа Эрланга (Agner Krarup Erlang, 1878–1929), широко известного своими исследованиями в сфере телекоммуникаций. В честь А. Эрланга названа также мера телекоммуникационного трафика в телефонии.

Центральным понятием Erlang является процесс, который реализован средствами языка и не требует соответствующего функционала ОС. В Erlang процессы очень легковесны, сравнимы с вызовом функций в императивных языках, поэтому работающие программы без труда могут запускать их тысячами и даже миллионами¹. Язык содержит три основные примитивные операции: создание процесса, отправка сообщения и получение процессом сообщения из собственного почтового ящика. Являясь функциональным языком, Erlang не содержит операции присваивания, что позволяет избавиться от необходимости применения сложных механизмов синхронизации процессов в многопоточном программировании (семафоров, мьютексов, событий).

Использование графических процессоров привело к созданию многочисленных графических API (*OpenGL*, *Direct3D*) для обработки данных на GPU, которые, по сути, имеют массово-параллельную архитектуру. Программа обработки изображения пишется на двух языках – на традиционном, например C++, и на языке для шейдеров². Шейдеры являются, в некотором смысле параллельными программами, но поскольку параллельная обработка точек не требует взаимодействия нитей, то API не предоставляет никакого механизма синхронизации и коллективных операций.

В настоящее время получила распространение предложенная компанией Nvidia технология *CUDA* (Compute Unified Device Architecture), которая вышла за рамки GAPI и предназначена для программирования массивно-параллельных вычислительных устройств (GPU компании Nvidia, начиная с серии GeForce8, семейства Tesla и пр.). Основная идея технологии заключается в том, что GPU является массивно-параллельным «сопроцессором» к CPU, обладающим собственной памятью. Последовательная часть программы, написанной на CUDA, выполняется на CPU, а параллельная – на GPU в виде набора одновременно выполняющихся нитей. Нити GPU обладают крайне небольшой стоимостью создания, управления и уничтожения (контекст нити минимален, все регистры распределены заранее). Реализация синхронизации нитей является некоторым компромиссом между необходимостью взаимодействия нитей и стоимостью такого взаимодействия. Взаимодействовать могут только нити одного блока сетки нитей путем либо разделения памяти, либо барьерной синхронизацией.

Параллельность «приобрели» не только языки программирования, но и системы инженерного анализа, предназначенные для 2D- и 3D-моделиро-

¹ Существуют эксперименты, демонстрирующие *сотни тысяч* созданных и уничтоженных процессов в секунду (<http://www.lshift.net/blog/2006/09/10/erlang-processes-vs-java-threads>).

² Шейдер – подпрограмма создания эффектов подсветки поверхности объекта, построения теней, текстур и прочих графических спецэффектов.

вания сложных (в том числе турбулентных) течений жидкости и газа, деформаций твердых тел, многофазных явлений. Как правило, такие системы включают все этапы численного моделирования: построение и адаптацию сеток, визуализацию постановки дифференциальной задачи и обработку результатов, выбор параметров численных методов. Например, популярный пакет конечноэлементного решения стационарных и нестационарных пространственных задач механики деформируемого твёрдого тела и механики конструкций *ANSYS* выпускается с возможностью автоматического распараллеливания расчетов. Другой пакет компании *ANSYS* (поглотившей конкурента) – *ANSYS FLUENT* – предназначен для моделирования сложных течений жидкостей и газов с широким диапазоном свойств. Интересна также российская разработка – программный комплекс по вычислительной гидродинамике *FlowVision*.

Приведенный краткий обзор языков, библиотек и систем параллельного программирования далеко не полон. С одной стороны, это дает основание надеяться, что для решения любой задачи найдется эффективный языковой инструментарий, а с другой стороны, свидетельствует об отсутствии единственного оптимального решения.

Параллельное программирование – это та область программирования, которая еще долго будет искусством, а не ремеслом.

1.5. Парадигмы параллельных приложений

Существует ряд схем взаимодействия самостоятельных частей параллельного приложения. Кратко опишем основные.

Итеративный параллелизм используется для реализации параллелизма в итеративной программе (чаще всего в циклах). Такой параллелизм характерен для распараллеливания по данным в согласованных параллельных вычислениях.

Рекурсивный параллелизм используется в программах с одной или несколькими рекурсивными процедурами, вызов которых независим. Каждый рекурсивный вызов порождает один или несколько новых процессов, которые независимо работают над решением задачи. В рамках такой парадигмы часто реализуются технологии «разделяй и властвуй» или «перебор с возвратом».

Итеративный и рекурсивный параллелизм основан на приемах, известных в последовательном программировании. Следующие схемы взаимодействия характерны именно для параллельных программ.

«*Производители и потребители*» – модель взаимодействия неравноправных процессов по поводу общих данных. Одни процессы «производят»

данные, другие – их «потребляют». Часто такие процессы организуются в *конвейер*, через который проходит информация. Каждый процесс конвейера потребляет выход своего предшественника и производит входные данные для своего последователя. Другой распространенный способ организации потоков – древовидная структура, на ней основан, в частности, принцип *дихотомии*.

«*Клиенты и серверы*» – наиболее распространенная модель взаимодействия процессов в распределенных системах. Клиентский процесс запрашивает (возможно, неоднократно) данные у сервера и ожидает ответа, затем использует полученные данные по своему усмотрению. Серверный процесс ожидает запроса от клиента, далее в соответствии с поступившим запросом обрабатывает данные и возвращает запросившему их процессу-клиенту. Таким образом, в отличие от предыдущего случая, между клиентом и сервером необходимо установить двустороннюю связь. Сервер может быть реализован как одиночный процесс, обслуживающий одновременно несколько клиентских процессов. Сервер может быть многопоточной программой, каждый поток которой обслуживает своего клиента. Если клиент и сервер выполняются на одном компьютере, то они представляют собой параллельное программное обобщение процедур: сервер исполняет роль процедуры, а клиент ее вызывает. Однако если коды клиента и сервера разнесены в пространстве, то для синхронизации используются специальные технологии, такие как удаленный вызов процедур или *рандеву*.

«*Управляющий и рабочие*» – модель организации вычислений, при которой существует поток, координирующий работу всех остальных потоков. Как правило, управляющий поток распределяет данные, собирает и анализирует результаты. Эта парадигма часто применяется в задачах оптимизации и статистической обработки информации, при обработке изображений и других научных вычислениях с итеративными алгоритмами.

«*Взаимодействующие равные*» – модель, в которой исключен не занимающийся непосредственными вычислениями управляющий поток. Распределение работ в таком приложении либо фиксировано заранее, либо динамически определяется во время выполнения. Одним из распространенных способов динамического распределения работ при создании программ для ВС с общей памятью является *портфель задач*. Портфель задач, как правило, реализуется с помощью разделяемой переменной, доступ к которой в один момент времени имеет только один процесс. Если же память ВС распределенная, то такая схема распределения работ превращается в схему «управляющий – рабочий», поскольку портфель задач оформляется отдельным процессом. Основными примерами в этой области являются научные вычисления с итеративными алгоритмами и системы, требующие децентрализованного принятия решений.

Нотация для определения параллельных вычислений

Множество чтения операции – это множество переменных, которые в ходе операции читаются, но не изменяются. *Множество записи операции* – это множество переменных, которые в ходе операции изменяются (и, возможно, читаются).

Две операции называются *независимыми*, если их множества записи не пересекаются.

Две операции могут выполняться параллельно, если они независимы, т. е. процессы всегда могут безопасно читать переменные, которые не изменяются. Однако двум процессам небезопасно выполнять запись в общую переменную или одному процессу читать переменную, которую изменяет другой процесс.

Для иллюстрации обсуждаемых здесь и далее алгоритмов введем общую нотацию, подобную рассмотренной в [48]. Особенности ее конкретных реализаций с помощью функций WinAPI, технологии OpenMP и библиотеки MPI описаны в соответствующих главах.

Для определения параллельного выполнения нескольких различных потоков команд примем следующую нотацию (оператор `par` – от «parallel»):

```
par
{ ... ;} // группа операторов 1
\\      // разделитель различных ветвей оператора
{ ... ;} // группа операторов 2
endpar
```

Для независимого выполнения тела цикла для каждого значения счетчика оператор `par` будем записывать в следующем виде:

```
par for (i=0;i<n;i++){
... // параллельное выполнение тела цикла для каждого i
}
```

Параллельно потоки команд или тело цикла могут выполняться как реально, так и только теоретически, что зависит от числа процессоров при реализации.

Другой способ определить параллельные вычисления – использовать декларацию `thread` (оператор `par`, выполняемый как фоновый):

```
thread t_name{
... // код процесса, который может выполняться
    // параллельно с другими процессами
}
```

Можно также определить группу процессов, выполняющую совместно (но не синхронно!) один и тот же код:

```
thread t_name (i=0;i<P) {
  ... // параллельное выполнение каждого из n процессов
}
```

Когда встречается данная команда, создается новый процесс (группа процессов), который начинает выполняться, возможно, параллельно. Предполагается, что при создании группы процессов существует механизм определения процессом размера группы P и своего номера i в группе, процессы в группе нумеруются подряд.

Еще раз подчеркнем, что в рамках данной нотации не обсуждается различие между потоком и процессом. Далее просто будем считать, что при моделировании систем с общей памятью процессы могут разделять переменные, т. е. возможно описание глобальных переменных, доступных всем процессам, а при моделировании с распределенной памятью процессы разделяют только каналы, обмен данными происходит с помощью примитивов обмена сообщениями `send` и `receive`.

Если за декларацией процесса `thread t_name(...) {...}` существуют операторы, то они выполняются параллельно с новыми процессами, операторы, следующие за телом декларации `par`, выполняются только после завершения всех процессов, порожденных в `par`, т. е. эта нотация предполагает в конце неявную синхронизацию *барьером*.

Основное отличие двух объявлений параллельного выполнения в том, что реализация декларации `thread t_name {...}` требует средств явного порождения процесса, а нотация `par` – наличия, например, опций компилятора, указывающего, что данный цикл должен выполняться параллельно. Последний подход реализован, в частности, в технологии OpenMP.

Рассмотрим несколько примеров параллельных алгоритмов, использующих описанные парадигмы [48]. При описании алгоритмов обратите внимание на нотацию определения основных конструкций последовательных языков. Особо отмечаются только неочевидные обозначения.

Итеративный параллелизм: произведение матриц

Одной из классических задач, используемых обычно для демонстрации параллелизма по данным в согласованных параллельных вычислениях, является перемножение матриц.

Задача 1.1. Даны матрицы A и B размерностью $n \times n$. Вычислить произведение матриц, поместив результат в матрицу C размерностью $n \times n$.

Пусть n объявлена и инициализирована, пусть также исходные матрицы A , B и матрица результатов C являются разделяемыми, матрицы A и B уже инициализированы:


```
int n=10000 ; //объявление и инициализация n
double a[n][n], b[n][n], c[n][n];
... //инициализация матриц A и B
```

Для определения матрицы c необходимо вычислить n^2 промежуточных произведений. Текст последовательной программы очевиден:

```
for (i=0; i<n; i++){
    for (j=0; j<n; j++){
        c[i][j] = 0.0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}
```

Поскольку множества записи внутренних произведений не пересекаются, их можно выполнять параллельно. Такие алгоритмы обычно называют *алгоритмами с массовым параллелизмом*. Возможны варианты, когда параллельно вычисляются результирующие строки, столбцы или группы строк и столбцов. Например, если выполнять тело цикла по i для каждого значения счетчика параллельно, то параллельно будут насчитываться строки произведения. Поменяв циклы по i и по j местами (это возможно из-за независимости операций тела вложенных циклов по i и по j) и выполнив параллельно тело цикла для каждого значения счетчика j , получим вариант программы, насчитывающий параллельно столбцы. Более того, можно параллельно насчитывать каждый элемент результирующей матрицы.

Пример 1.1 демонстрирует параллельное вычисление произведения матриц по строкам. Выполнение оператора `par` здесь порождает столько потоков, сколько строк в матрице A . Распределение потоков по процессорам зависит от среды выполнения и реализации.

Пример 1.1. Вычисление произведения матриц по строкам

```
int n= ; //объявление и инициализация n
double a[n][n], b[n][n], c[n][n];
... //инициализация массивов a и b
par for (i=0; i<n; i++){ // параллельное вычисление строк
    for (j=0; j<n; j++){
        c[i][j] = 0.0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}
```

Предположим, что процессы в системе могут разделять переменные. Разделим матрицу на полосы строк, для каждой полосы создадим свой рабочий процесс. Пусть число запускаемых для решения задачи процессов известно системе и равно P ($P < n$, n нацело делится на P), все процессы выполняют один код (пример 1.2). Поскольку множества записи процессов не пересекаются, то процессы корректно вычислят произведение матриц.

Пример 1.2. Вычисление произведения матриц по полосам

```
int n= ; // объявление и инициализация n
double a[n][n], b[n][n], c[n][n];
... // инициализация массивов a и b
thread worker (w=0; w<P){ //полосы вычисляются параллельно
//Номер первой строки полосы для процесса w
int first = w*n/P;
//Номер последней строки в полосе для процесса w
int last = first + n/P - 1;
for (i=first; i<=last; i++)
for (j=0; j<n; j++){
    c[i][j] = 0.0;
    for (k=0; k<n; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
}
```

Управляющий и рабочие: распределенное умножение матриц

При обсуждении итеративного параллелизма рассматривался алгоритм параллельного умножения матриц с помощью процессов, которые разделяют общие переменные. На машинах с распределенной памятью любая переменная должна быть локальной для некоторого процесса и может быть доступна только этому процессу. Для обмена информацией процессы используют передачу сообщений.

Рассмотрим два способа реализации алгоритма умножения матриц для архитектур с распределенной памятью. В первом случае процессы будут неравноправны – выделен управляющий процесс и пул *независимых* процессов-вычислителей. Во втором случае имеет место взаимодействие процессов-вычислителей между собой, которое обеспечивается круговым конвейером (рис. 1.12–1.13). Кроме того, каждый вычислитель взаимодействует с управляющим процессом.

Вновь рассмотрим задачу 1.1.

Пусть n объявлено и инициализировано. Пусть также имеется n рабочих процессов, каждый из которых вычисляет одну строку результирующей матрицы. Для этого каждый рабочий процесс должен располагать

одной строкой матрицы A и *всей* матрицей B . Пусть также существует управляющий процесс, который посылает каждому рабочему процессу необходимые данные и собирает результаты расчетов, т. е. строки матрицы C . Ниже приведены алгоритмы обоих процессов (примеры 1.3 и 1.4 соответственно).

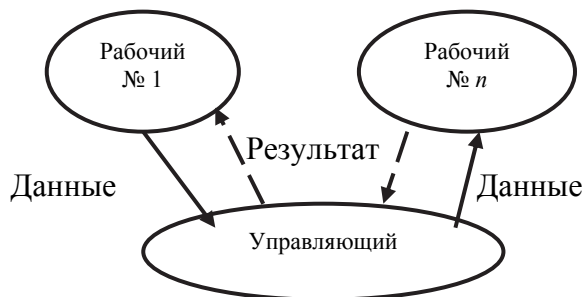


Рис. 1.12. Взаимодействие «управляющий – рабочий»

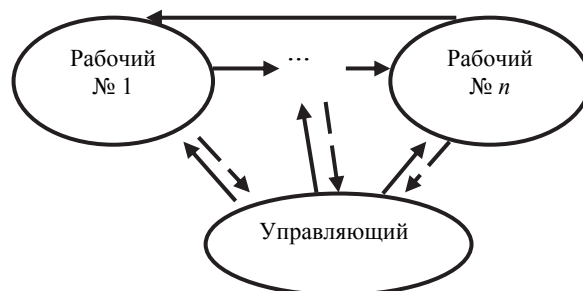


Рис. 1.13. Круговой конвейер

Операторы `send` и `receive` – это нотация примитивов передачи сообщений. Операция `send` упаковывает сообщение и пересылает его другому процессу; операция `receive` ожидает сообщение от другого процесса, получает его, распаковывает и сохраняет в локальных переменных. О синхронизации этих примитивов мы подробно поговорим в гл. 4 и 6.

Пример 1.3. Умножение матриц для ВС с распределенной памятью. Взаимодействие «управляющий – рабочий». Код рабочего процесса

```
thread worker (i=0; i<n){
    double a[n];    // строка i матрицы A
    double b[n][n]; // вся матрица B
    double c[n];    // строка i матрицы C
    receive значения элементов массива a, соответствующих i-й
    строке матрицы A;
    receive значения массива b;
    for (j=0; j<n; j++) {
        c[j] = 0.0;
        for (k=0; k<n; k++)
            c[j] = c[j] + a[k] * b[k][j];
    }
    send вектор-результат к управляющему процессу;
}
```

Пример 1.4. Умножение матриц для ВС с распределенной памятью. Взаимодействие «управляющий – рабочий». Код управляющего процесса

```
thread manager {
  double a[n][n]; // исходная матрица A
  double b[n][n]; // исходная матрица B
  double c[n][n]; // результирующая матрица C
  инициализировать a и b;
  for (i=0; i<n; i++) {
    send строку i массива a процессу worker[i];
    send весь массив b процессу worker[i];
  }
  for (i=0; i<n; i++)
    receive строку i массива c от процесса worker[i];
  вывести результат, который теперь в массиве c;
}
```

Пример 1.5. Умножение матриц для ВС с распределенной памятью. Взаимодействие управляющий-рабочий, круговой конвейер. Код рабочего процесса

```
thread worker (w=0; w<n){
  double a[n]; // одна строка матрицы A
  double b[n]; // один столбец матрицы B
  double c[n]; // одна строка матрицы C
  double sum = 0.0; // для промежуточных произведений
  int next_col = w; // следующий столбец результатов,
                    // сначала равен номеру процесса w

  receive строку w массива a и столбец w массива b
    от управляющего процесса;

  // вычислить c[w,w] = a[w,*] * b[:,w]
  for (k=0; k<n; k++)
    sum = sum + a[k] * b[k];
  c[next_col] = sum;

  // пустить по кругу столбцы и вычислить остальные c[j,*]
  for (j=1; j<n; j++) {
    send мой столбец матрицы b следующему процессу;
    receive новый столбец матрицы b от предыдущего процесса;
    sum = 0.0;
    for (k=0; k<n; k++)
      sum = sum + a[k] * b[k];
    if (next_col == 0) next_col = n - 1;
    else next_col = next_col - 1;
    c[next_col] = sum;
  }
  send вектор-результат управляющему процессу;
}
```

Пусть теперь рабочие процессы изначально имеют по i -й строке матрицы A и по j -му столбцу матрицы B . Все рабочие процессы соединены в круговой конвейер, по которому циркулируют столбцы матрицы B (рис. 1.13). Код рабочего процесса в этом случае приведен в примере 1.5, порядок заполнения элементов результирующей матрицы изображен на рис. 1.14.

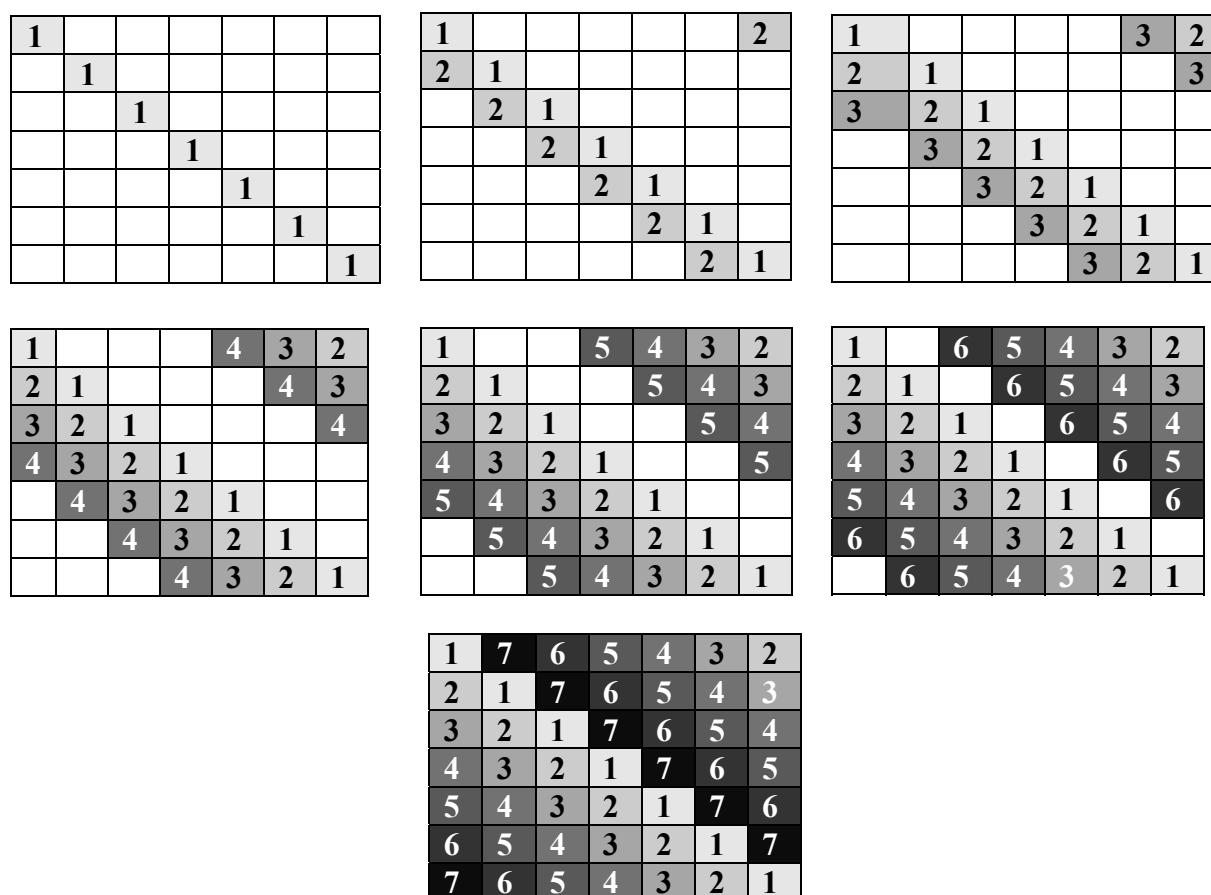


Рис. 1.14. Вычисление элементов результирующей матрицы (слева направо и сверху вниз показан порядок вычисления элементов матрицы согласно циклу по j примера 1.5)

Для запуска кругового конвейера требуется процесс-инициатор вычислений. Его код будет отличен от всех остальных. Этот процесс может быть управляющим и заниматься только первоначальной рассылкой и окончательной сборкой результатов, тогда его код будет такой же, как и в предыдущем случае (пример 1.4).

В чем же различие двух алгоритмов? В первом случае (примеры 1.3–1.4, рис. 1.12) значения матрицы B дублируются в памяти каждого рабочего процесса. В алгоритме из примеров 1.4–1.5 (рис. 1.13) в каждый момент времени каждый процесс хранит только одну строку матрицы A и один

столбец матрицы В. Это снижает затраты на память процесса, но вторая программа выполняется дольше первой, поскольку на каждой итерации ее рабочий процесс должен отослать сообщение одному соседу и получить сообщение от другого. Это типичное противоречие между требуемым объемом памяти и временем выполнения.

Взаимодействующие равные: умножение матриц с помощью портфеля задач

В предыдущем пункте управляющий процесс не участвует непосредственно в вычислениях, поэтому иногда удобно от него избавиться, позволив рабочим процессам самим распределять между собой задания или данные. Одним из общих подходов к динамическому распределению задач (данных) по процессам (потокам) является *портфель задач*.

В этом подходе явно прослеживается единство параллелизма данных и задач. Вычислительная задача делится на конечное число подзадач, являющихся независимыми единицами работы. Каждая подзадача может выполнить однотипные действия над разными данными (яркий пример — получение элемента произведения матриц). Подзадачи нумеруются, т. е. каждому номеру соответствует свой набор данных, точнее над множеством номеров задач определяется функция, которая однозначно отражает номер задачи на соответствующий ему набор данных. Создается переменная или, если нумерация сложная, то структура для хранения номера задачи, которую следует выполнять следующей (например, номер столбца, в котором надо посчитать элементы произведения). Каждый поток независимо выполняет следующую цепочку действий: 1) обращается к портфелю задач для выяснения текущего номера задачи; 2) увеличивает портфель задач на единицу; 3) выполняет задачу, используя соответствующие данные; 4) обращается к портфелю задач для получения следующего номера задачи. Должен быть предусмотрен механизм остановки процессов при исчерпывании всего множества задач. Общий алгоритм отображен в примере 1.6.

Пример 1.6. Портфель задач

```
int NextProblem = 0; //портфель задач

thread worker (w=0; w<n) {
    Получить задачу из портфеля;
    while (задача существует) {
        Выполнить задачу;
        Получить задачу из портфеля;
    }
}
```

Таким образом, процессы (потoki) работают независимо, каждый со своей скоростью, синхронизация происходит с помощью портфеля задач, в качестве которого удобно использовать глобальную переменную.

Проблема реализации этого алгоритма в том, что доступ к портфелю задач должен быть безопасным. Оператор «получить задачу из портфеля», как правило, предполагает неделимое выполнение минимум двух операций: считывание текущего значения переменной `NextProblem` и увеличение ее на единицу. Если между этими двумя действиями работа выполняющего их потока прервется, то некоторую задачу, возможно, обработают несколько потоков.

Рассмотрим решение задачи 1.1 об умножении матриц с помощью описанного подхода.

В качестве подзадачи выберем вычисление строки результирующей матрицы, тогда портфель задач – это просто счетчик строк, реализованный как глобальная переменная. Операция получения задачи из портфеля – считывание текущего значения счетчика с последующим его увеличением на единицу. Реализация должна предоставлять механизм, гарантирующий выполнение этих действий атомарно. Признаком окончания вычисления результирующей матрицы является равенство портфеля задач размерности матриц n . Код программы демонстрирует пример 1.7.

Пример 1.7. Умножение матриц с помощью портфеля задач

```
int n = ; //объявление и инициализация n
double a[n][n], b[n][n], c[n][n];
Инициализация массивов a и b
int NextRow = 0; //портфель задач
thread worker (w=0; w<n){
    int row_current;

    //Получить задачу из портфеля, неделимо выполнив операторы
    {row_current = NextRow; NextRow++;}

    while (row_current < n) {
        //Вычисление строки матрицы C
        for (j=0; j<n; j++) {
            c[row_current][j] = 0.0;
            for (k=0; k<n; k++)
                c[row_current][j]=c[row_current][j]
                    +a[row_current][k]*b[k][j];
        }

        //Получить задачу из портфеля, неделимо выполнив операторы
        {row_current = NextRow; NextRow++;}
    }
}
```

Можно выбрать в качестве подзадачи вычисление одного элемента результирующей матрицы с или ее столбца.

Отметим, что мы рассмотрели пять способов параллельной реализации алгоритма умножения матриц. Все изученные реализации различаются механизмами синхронизации параллельных процессов и способами распределения данных по процессам. В табл. 1.3 отражены характерные особенности рассмотренных алгоритмов. Следует также отметить, что приведенными примерами не исчерпываются все возможные способы распараллеливания этого простейшего алгоритма. Выбор наиболее подходящего из них тоже является отдельной весьма сложной проблемой, обсуждение которой мы еще раз немного затронем в гл. 5.

Таблица 1.3

Алгоритмы параллельного умножения матриц				
ВС с общей памятью			ВС с распределенной памятью	
Итеративный параллелизм		Взаимодействующие равные	Управляющий – рабочий	
Параллелизм по строкам (столбцам, элементам)	Параллелизм по блокам строк (столбцов, элементов): статическое распределение данных	Портфель задач: динамическое распределение данных	Управляющий – рабочий	Круговой конвейер

Рекурсивный параллелизм: адаптивная квадратура

Программа считается рекурсивной, если она содержит процедуры, прямо или косвенно вызывающие сами себя. Рекурсивные программы можно преобразовать в итеративные, и наоборот. Рекурсивную программу можно выполнять как множество параллельных процессов, если она содержит несколько независимых рекурсивных вызовов.

Два вызова процедуры независимы, если их множества записи не пересекаются. Это условие выполняется, если:

- процедура не обращается к глобальным переменным или только читает их;
- аргументы и результирующие переменные суть различные переменные.

Для иллюстрации рассмотрим алгоритм вычисления определенного интеграла, носящий название адаптивной квадратуры.

Задача 1.2. Дана непрерывная на отрезке $[a, b]$ функция $f(x)$. Требуется приближенно вычислить интеграл

$$\int_a^b f(x)dx. \quad (1.1)$$

Существует, по крайней мере, два принципиально различных способа численной аппроксимации значения интеграла.

Для реализации первого из них следует разделить интервал $[a, b]$ на фиксированное число отрезков, а затем аппроксимировать интеграл на каждом из отрезков с помощью некоторой квадратурной формулы (например, трапеций или Симпсона). Такая программа будет итеративной, причем она должна вычислять сумму n чисел. Идея эффективной параллельной организации подобных вычислений рассмотрена при обсуждении парадигмы «потребитель–производитель».

Второй способ обычно носит название *адаптивной квадратуры* и использует метод последовательных приближений с заданной точностью ε . При его реализации сначала вычисляют аппроксимацию интеграла (1.1) по двум точкам a и b (начальное приближение). Затем делят отрезок $[a, b]$ на два $[a, m]$ и $[m, b]$, где m в простейшем случае может быть серединой начального отрезка. Вычисляют квадратуру как сумму аппроксимаций интеграла на каждом отрезке. Если разность первой и второй квадратур меньше заданной точности ε , то за приближенное значение интеграла принимают вычисленную сумму. Иначе задача вычисления интеграла (1.1) делится на две подзадачи: вычисления интегралов от a до m и от m до b . Процесс повторяется независимо для каждого нового отрезка. Такой алгоритм удачно представить рекурсивной процедурой.

Предположим, что подынтегральная функция вычисляется вызовом функции `double f(double x);`

Определим рекурсивную функцию `q_integral()`, реализующую описанный алгоритм (пример 1.8). Аргументами `q_integral()` являются значения левого `left` и правого `right` концов отрезка, значения функций `f_left`, `f_right` в этих точках и текущее значение интеграла `intgrl_now`.

Пример 1.8. Функция рекурсивного вычисления значения определенного интеграла

```
double q_integral (double left, double right,
                  double f_left, double f_right,
                  double intgrl_now){
    double mid = (left + right) / 2;
    double f_mid = f(mid);
    //Аппроксимация по левому отрезку
    double l_integral = (f_left + f_mid) * (mid - left) / 2;
    //Аппроксимация по правому отрезку
    double r_integral = (f_mid + f_right) * (right - mid) / 2;
    if (abs ((l_integral + r_integral) - intgrl_now) > EPS){
        //Рекурсия для интегрирования обоих значений
        l_integral = q_integral(left, mid, f_left, f_mid, l_integral);
        r_integral =
            q_integral(mid, right, f_mid, f_right, r_integral);
    }
    return (l_integral + r_integral);
}
```

Интеграл (1.1) приближенно вычисляется вызовом функции:

```
area = q_integral(a,b,f(a),f(b),(f(a)+f(b))*(b-a)/2);
```

Легко заметить, что рекурсивные вызовы функции `q_integral()` независимы при условии, что вычисление функции $f(x)$ не дает побочных эффектов. Заменим рекурсивные вызовы в теле функции на оператор `par` с рекурсивным параллелизмом (разделитель `\\` отделяет процессы, которые допустимо выполнять независимо):

```
par
  l_integral = q_integral(left,mid,f_left,f_mid,l_integral);
  \\
  r_integral = q_integral(mid,right,f_mid,f_right,r_integral);
end par
```

Оператор `par` не заканчивается до тех пор, пока не будут завершены оба вызова функций. Таким образом, значения переменных `l_integral` и `r_integral` вычисляются до того, как функция `q_integral()` возвращает их сумму. Неприятность может возникнуть в накоплении большого количества параллельно выполняющихся процессов при большой глубине рекурсии, ведь каждый оператор `par` создает два независимых процесса. Решение этой проблемы может состоять, например, в переключении с параллельных рекурсивных вызовов на последовательные при достижении некоторой критической глубины рекурсии.

Производители и потребители: каналы ОС Unix

Процесс-производитель выполняет вычисления и выводит поток результатов. *Процесс-потребитель* вводит поток значений и анализирует его. Многие программы в той или иной степени являются производителями и/или потребителями.

Интересен случай, когда производители и потребители объединены в *конвейер* – последовательность процессов, в которой каждый потребляет данные предшественника и предоставляет данные для последующего процесса (рис. 1.16). Классическим примером являются конвейеры в ОС Unix. Рассмотрим взаимодействие команд:

```
$ps -Af | grep mc
```

Вертикальная черта «|» между командами обозначает конвейер, т. е. выход одной команды переназначается на вход другой команды, таким образом, в приведенном выше примере результат выполнения `ps -Af` будет передан команде `grep`, которая произведет трансформацию входного потока в соответствии с маской (в данном случае на выходе будут все строки, содержащие подстроку «mc»).

Обычно с процессом связан стандартный поток ввода `stdin` и стандартный поток вывода `stdout`. Эти потоки могут быть связаны с файлами особого типа – каналами. *Канал* – это буфер (очередь типа FIFO) между процессом-производителем и процессом-потребителем. Он содержит связанную последовательность символов, причем процесс-производитель записывает данные в конец очереди, а процесс-потребитель читает данные из ее начала, при этом символы удаляются. В общем случае канал – это ограниченный буфер, поэтому производитель при необходимости ожидает, пока в буфере появится свободное место для очередной порции данных, а потребитель при необходимости ждет появления в буфере данных.

При общей памяти буферы реализуются с помощью разделяемых переменных и примитивов синхронизации (флагов, семафоров, монитора). Разбор возможного алгоритма решения этой задачи на общей памяти обсуждается в гл. 2 (задача о кольцевом буфере). При распределенных процессах необходима реализация канала взаимодействия с помощью примитивов пересылки сообщений (`send` и `receive`).

Между производителем и потребителем существует однонаправленный поток информации. Такой вид межпроцессного взаимодействия не имеет аналогов в последовательном программировании, поскольку в последовательной программе только один поток управления, в то время как производители и потребители – независимые процессы со своими потоками управления и собственными скоростями выполнения.

Другим способом организации потоков потребителей-производителей являются различные древовидные структуры. Самая распространенная из них – организация процесса вычислений *методом сдваивания (дихотомия)*. С помощью этого мощного подхода удобно распараллеливать алгоритмы нахождения сумм элементов массива, его максимального элемента и других групповых функций от массивов, а также вычисления по различным рекуррентным формулам. Такие задачи часто встречаются как составляющие алгоритмов (например, описанный выше первый способ нахождения приближенного значения интеграла). Основную идею метода дихотомии удобно продемонстрировать при распараллеливании алгоритма сложения n чисел. Поскольку в большинстве реальных программ (почему не во всех?) нет существенной разницы, в каком порядке складывать числа, сложение производят по следующей схеме. Сначала находят сумму пар соседних элементов массива: $a[1]+a[2]$, $a[3]+a[4]$, $a[5]+a[6]$ и т. д. Это можно делать одновременно. На следующем шаге попарно суммируют значения, полученные на предыдущем шаге, и так до получения окончательного результата (рис. 1.15).

Отметим, что алгоритм сложения сдваиванием является оптимальным по количеству операций, которые можно выполнять параллельно. Это

означает, что при наличии необходимого числа сумматоров вычисление результата произойдет за минимальное число шагов. С другой стороны, эффективность такого алгоритма сомнительна – на каждом параллельном шаге количество сумматоров, занятых в вычислениях, уменьшается вдвое, т. е. большую часть времени некоторое число сумматоров простаивают. К обсуждению этой проблемы мы вернемся в гл. 5.

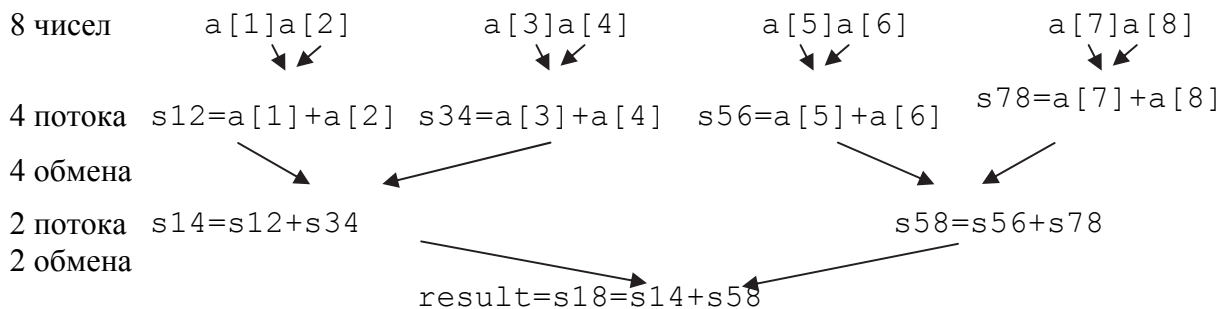


Рис. 1.15. Пример нахождения суммы восьми чисел методом сдваивания

Клиенты и серверы: сервер баз данных

Еще одной типичной схемой взаимодействия процессов является парадигма клиент-сервер. *Процесс-клиент* запрашивает сервис, затем ожидает результата обработки запроса. *Процесс-сервер* многократно ожидает запрос, обрабатывает его, отправляет ответ. Таким образом, между процессами существует двунаправленный поток информации (рис. 1.17).

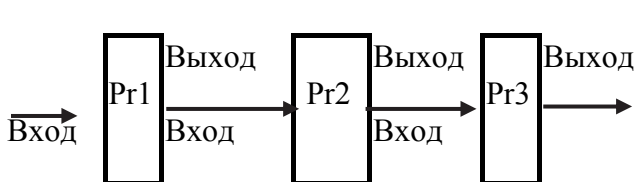


Рис. 1.16. Конвейер

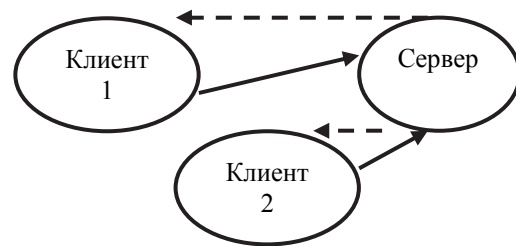


Рис. 1.17. Клиенты и сервер

Взаимодействие клиентов и сервера аналогично отношению между программой, вызывающей процедуру, и самой вызываемой процедурой в последовательном программировании. Более того, как процедура обычно вызывается из нескольких мест основной программы, так же и у сервера существует много клиентов. Запросы каждого клиента должны обрабатываться независимо, соответственно параллельно может обрабатываться несколько запросов. Аналогично одновременно может быть активно несколько вызовов одной и той же процедуры.

Взаимодействие «клиент – сервер» чрезвычайно распространено, оно предпочтительно в ядре ОС, объектно ориентированных системах, сетевых приложениях, СУБД.

Кратко опишем основную проблему программирования таких приложений. Пусть имеется модуль сервера базы данных (БД), обеспечивающий две операции работы с таблицами БД: read (читать) и write (изменять данные). Когда процесс-клиент хочет получить доступ к таблицам, он обращается к серверу БД с запросом на операцию чтения или записи в соответствующем модуле сервера. В реальной СУБД работа с данными идет транзакциями – осмысленными с точки зрения пользователя действиями, производимыми над данными. Транзакция должна гарантировать непротиворечивое состояние данных после своего окончания, в то время как в ходе транзакции согласованность данных может временно нарушаться.

В системе с разделяемой памятью сервер реализуется набором подпрограмм, а взаимодействие между сервером и клиентами реализуется вызовом процедуры. В распределенной системе клиенты и сервер разнесены в пространстве и общаются посредством передачи сообщений или вызовом удаленных процедур.

В любом случае, поскольку БД – разделяемый ресурс, то важно, чтобы запись в таблицу велась в монопольном режиме (проблема пропавших изменений), в то время как читать файл могут одновременно несколько процессов. Конфликты возникнут и тогда, когда процесс попытается записать данные в таблицу, которую читает другой процесс (проблема несогласованных данных), или процесс попытается прочитать данные из таблицы, в которую пишет другой процесс (проблема промежуточных данных). Механизм транзакций обеспечивается примитивами синхронизации (флагами, семафорами, монитором). Разбор возможного алгоритма решения этой задачи обсуждается в гл. 2 (задача о читателях и писателях).

Мы рассмотрели основные методы, используемые при написании параллельных программ. Даже при разборе простейших примеров становится ясно, что существует два принципиально разных подхода к программированию синхронизации задач в зависимости от того, разделяют ли потоки (процессы) их выполняющие переменные или данные разнесены в пространстве.

Если память общая, то возникает борьба за разделяемый ресурс (в том числе переменные). Проблема решается с помощью *флагов, семафоров* и *мониторов*. Если же память у процессов распределенная, то им приходится *обмениваться сообщениями* при использовании общих данных. Общие положения этих подходов, а также некоторые их языковые и библиотечные реализации и являются предметом дальнейшего рассмотрения.

Контрольные вопросы и задания

1. Приведите примеры задач, решаемых с использованием высокопроизводительных параллельных вычислительных систем.
2. Чем объясняется разнообразие архитектур параллельных вычислительных систем?
3. Назовите основные виды архитектур параллельных вычислительных систем. Ответ проиллюстрируйте примерами.
4. В чем проявляется специфика векторно-конвейерных архитектур?
5. Назовите основные особенности параллельных вычислительных систем с общей памятью.
6. Какова специфика параллельных вычислительных систем с распределенной памятью?
7. Каковы основные критерии классификации Флинна? Назовите основные группы вычислительных систем, соответствующие этой классификации.
8. Какие виды вычислительных систем относятся к SISD-, SIMD-, MISD- и MIMD-архитектурам? Ответ проиллюстрируйте примерами.
9. Дайте обзор архитектур первых пяти самых высокопроизводительных компьютеров из текущего списка top500.
10. Чем обуславливается разнообразие методов написания параллельных программ?
11. Приведите примеры задач, обладающих явным параллелизмом по данным.
12. Приведите примеры задач, обладающих явным параллелизмом по задачам.
13. Перечислите три основные составляющие процесса проектирования параллельных приложений. Ответ проиллюстрируйте примером.
14. Какова может быть логика разбиения прикладной задачи на части?
15. Перечислите уровни параллелизма, проиллюстрируйте ответ примерами.
16. Перечислите проблемы синхронизации отдельных частей целого приложения. Ответ проиллюстрируйте примерами.
17. Какие языки параллельного программирования, библиотеки и системы разработки параллельных программ вы знаете?
16. Сделайте обзор какой-либо системы, языка или библиотеки параллельного программирования.
17. Перечислите основные парадигмы параллелизма, используемые при построении параллельных программ.
19. Опишите сходства и различия нотации параллелизма операторных скобок `par ... endpar` и `thread`.

20. В чем специфика реализации алгоритма нахождения произведения матриц для SIMD-архитектуры?

21. Каковы могут быть варианты алгоритма нахождения произведения матриц для MIMD-архитектур?

22. В каком случае возможно применение рекурсивного параллелизма. Ответ проиллюстрируйте примерами.

23. Расскажите о проблемах, возникающих при использовании парадигмы «производитель – потребитель» на примере задачи организации обмена данными с помощью общего буфера в системах с общей памятью.

24. Расскажите о проблемах, возникающих при использовании парадигмы «клиент – сервер» на примере задачи чтения и записи в файл.

25. Какова специфика метода портфеля задач? Приведите пример.

26. В чем заключается метод организации потоков по принципу дихотомии? Приведите пример.

Задачи

1.1. Автобусный билет называется счастливым, если сумма трех первых цифр его номера равна сумме трех последних. Требуется подсчитать количество счастливых билетов в бобине (от 000000 до 999999). Напишите алгоритм решения задачи с помощью:

а) парадигмы «управляющий – рабочий» для машины с распределенной памятью;

б) портфеля задач для машины с общей памятью.

Сравните эти два подхода. Укажите места алгоритма, требующие синхронизации процессов.

1.2. Найти обратную матрицу для матрицы A . Входные данные: целое положительное число n , произвольная матрица A размерности $n \times n$. Написать алгоритм многопоточного приложения, если:

а) количество потоков является входным параметром, при этом размерность матриц может быть не кратна количеству потоков;

б) количество потоков заранее неизвестно, потоки в ходе выполнения алгоритма могут динамически создаваться и удаляться.

Используйте наиболее удобную схему взаимодействия потоков в каждом случае. Пояснить свой выбор. Укажите места алгоритма, требующие синхронизации процессов.

1.3. Написать параллельный алгоритм приближенного вычисления интеграла $\int_a^b f(x)dx$ с помощью квадратурной формулы метода трапеций.

При суммировании использовать принцип дихотомии. Укажите места алгоритма, требующие синхронизации процессов.

1.4. Изменить параллельный алгоритм приближенного вычисления интеграла $\int_a^b f(x)dx$ с помощью метода адаптивной квадратуры (пример

1.8) таким образом, чтобы при создании потоков больше порогового значения R дальнейшая рекурсия происходила последовательно. Укажите места алгоритма, требующие синхронизации процессов.

1.5. *Задача об экзамене.* Преподаватель принимает экзамен у N студентов, которые заходят группами по K человек ($N > KM$, $K > 2$, $M > 1$). Каждый студент получает билет от преподавателя, готовит ответ, передает его преподавателю, получает оценку и радостно сообщает оценку при выходе, если она положительная, или молча уходит, если получил двойку. Преподаватель раздает билеты каждой новой партии студентов, получает и просматривает ответы и сообщает каждому студенту его оценку. Новая группа студентов заходит только после того, как все студенты предыдущей группы получили оценки. Требуется описать алгоритмы, моделирующие действия потока-преподавателя и потоков-студентов. Какова схема взаимодействия потоков? Укажите критические участки алгоритма. Объясните, какие механизмы использовались для обеспечения корректной работы алгоритма при асинхронной работе потоков.

1.6. *Задача о производстве булавок.* В цехе по заточке булавок за смену необходимо обработать N булавок. Все необходимые операции осуществляются тремя рабочими. Первый из них берет булавку и проверяет ее на предмет кривизны. Если булавка не кривая, то рабочий передает ее своему напарнику. Напарник осуществляет собственно заточку и передает заточенную булавку третьему рабочему, который осуществляет контроль качества операции и складывает булавку в ящик для готовых булавок. Если на любом этапе обнаруживается брак, то булавка далее не обрабатывается, а складывается в ящик для бракованных булавок. Каждый рабочий в конце дня сообщает мастеру, сколько булавок он обработал. Мастер проверяет ящики, подводит баланс, сдает на переплавку брак, а на склад – готовую продукцию. Если мастер обнаруживает, что рабочие случайно потеряли хотя бы одну булавку, то все рабочие лишаются премии. Требуется: 1) придумать алгоритм для мастера по определению того, лишать рабочих премии или нет; 2) описать многопоточное приложение, моделирующее работу цеха в течение месяца. При решении использовать парадигмы «производитель – потребитель» и «управляющий – рабочий».

Глава 2 | ПРОБЛЕМЫ ПРОГРАММИРОВАНИЯ ДЛЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ С ОБЩЕЙ ПАМЯТЬЮ

В главе рассмотрены основы многопоточного программирования. Дается общее понятие потока, описываются способы его существования в системе. Подробно рассматривается проблема синхронизации потоков: взаимное исключение, барьеры, условное ожидание. Обсуждаются классические задачи и способы их решения, алгоритмы, основанные на активном ожидании, использовании семафоров. В конце кратко рассмотрено понятие монитора.

2.1. Анатомия потока

Под потоком понимается часть выполняемого кода в процессе ОС, которая должна быть регламентирована определенным образом. Затраты вычислительных ресурсов ОС, связанных с созданием потока, его поддержкой и управлением, значительно ниже по сравнению с аналогичными затратами для процесса. С точки зрения прикладного программиста основным отличием потока от процесса является то, что *потоки внутри одного процесса могут разделять переменные*.

Каждый процесс имеет основной, или первичный, поток, который запускается первым. В языке Си функция основного потока всегда носит имя `main()`. Каждый поток имеет свою последовательность инструкций, выполняется независимо от других потоков и, возможно, параллельно с ними.

Потоки, порожденные внутри одного процесса, равноправны и существуют в едином адресном пространстве, разделяя его ресурсы (дескрипторы файлов, глобальные переменные и т. д.). Потоки могут создавать, приостанавливать, запускать или даже завершать другие потоки в своем процессе. В то же время потоки — это выполняемые части программы, которые соревнуются между собой за использование процессора, памяти и других ресурсов. В многопроцессорной (многоядерной) среде потоки могут выполняться на разных процессорах (ядрах). В однопроцессорных системах они выполняются только за счет *переключения контекста*¹.

¹ Контекст переключается при поддержке операционной системой многозадачности. Каждая задача выполняется в течение выделенного интервала времени, после истечения которого (или после наступления некоторых других событий) текущая зада-

Для управления потоками используются соответствующие средства, которые доступны программисту через языковые конструкции, системные вызовы ОС или специально разработанные библиотеки. Например, библиотека потоков Pthread определяет объект атрибутов потока, инкапсулирующий свойства потока, к которым создатель объекта может получить доступ и модифицировать их. В ОС семейства Windows поток является объектом ядра. Каждый объект ядра – это блок памяти, выделенный ядром и доступный только ядру. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте (атрибуты потока). Объект ядра «поток» описывается атрибутами, общими для всех объектов ядра ОС (имя объекта, дескриптор защиты, счетчик числа пользователей), и атрибутами, характерными только для объекта «поток» (идентификатор процесса, которому он принадлежит, базовый приоритет, код завершения). Обсуждение особенностей потоков в ОС семейства Windows содержится в гл. 3.

Поток может пребывать в одном из четырех состояний [37, 42, 44]: готовности, выполнения, останова-ожидания, блокирования (рис. 2.1).

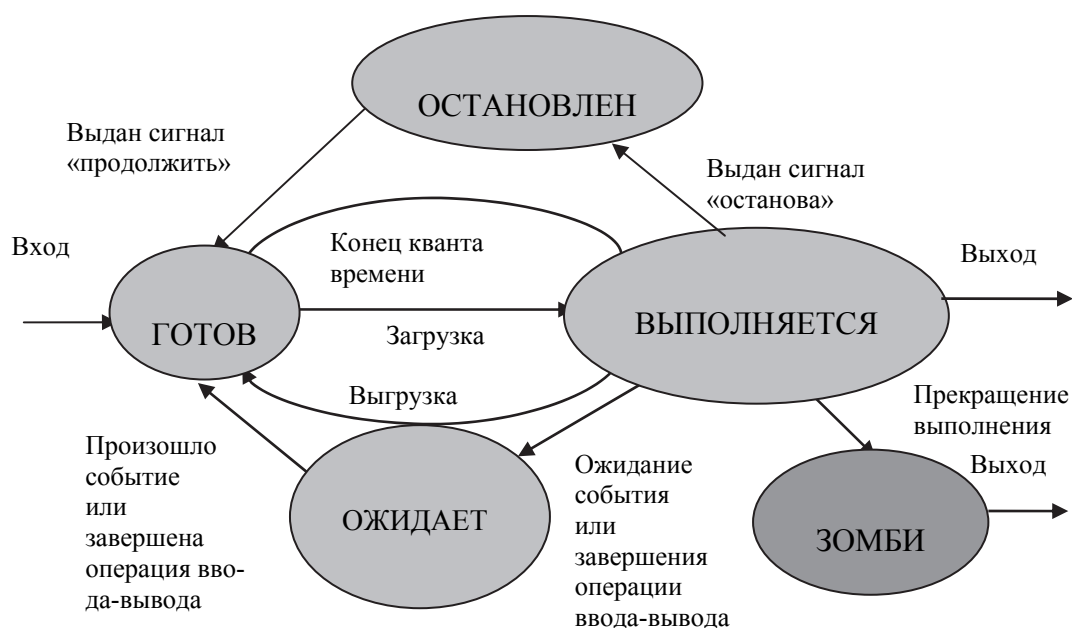


Рис. 2.1. Состояния потоков в средах Unix/Linux

Поток находится в состоянии *готовности*, когда он готов к выполнению. Все готовые к работе потоки помещаются в несколько очередей готовности согласно своим приоритетам.

ча снимается с процессора, а ему назначается другая, которая выбирается в соответствии с принятой в ОС дисциплиной планирования. Отметим, что при переключении контекста между потоками одного процесса затрачивается меньше ресурсов, чем при переключении контекста между процессами.

Поток переходит в состояние *выполнения*, когда он выбирается из очереди и назначается процессору. Поток снимается с процессора и помещается в очередь готовых потоков, если его квант времени истек или если перешел в состояние готовности поток с большим приоритетом.

Поток может оказаться не в состоянии продолжать выполнение до истечения кванта времени, например, сделав запрос на получение доступа к устройству ввода-вывода, вызвав системную функцию или ожидая освобождения переменной синхронизации. В этом случае поток «засыпает», перемещаясь в очередь *ждущих* потоков. После наступления ожидаемого события поток перемещается из очереди ждущих потоков в очередь готовых.

Выполняющийся поток может получить сигнал и перейти в состояние *останова*, которое принципиально отличается от состояния ожидания. Поток получает сигнал остановиться, если он находится в состоянии отладки или в системе возникает особая ситуация. Позже поток может быть разбужен или ликвидирован.

Если поток не является открепленным¹, то по завершении выполнения он переходит в специальное состояние «зомби», в котором уже не способен продолжать выполнение, не использует системных ресурсов, но не может покинуть систему, пока его родительский поток не извлечет статус его завершения.

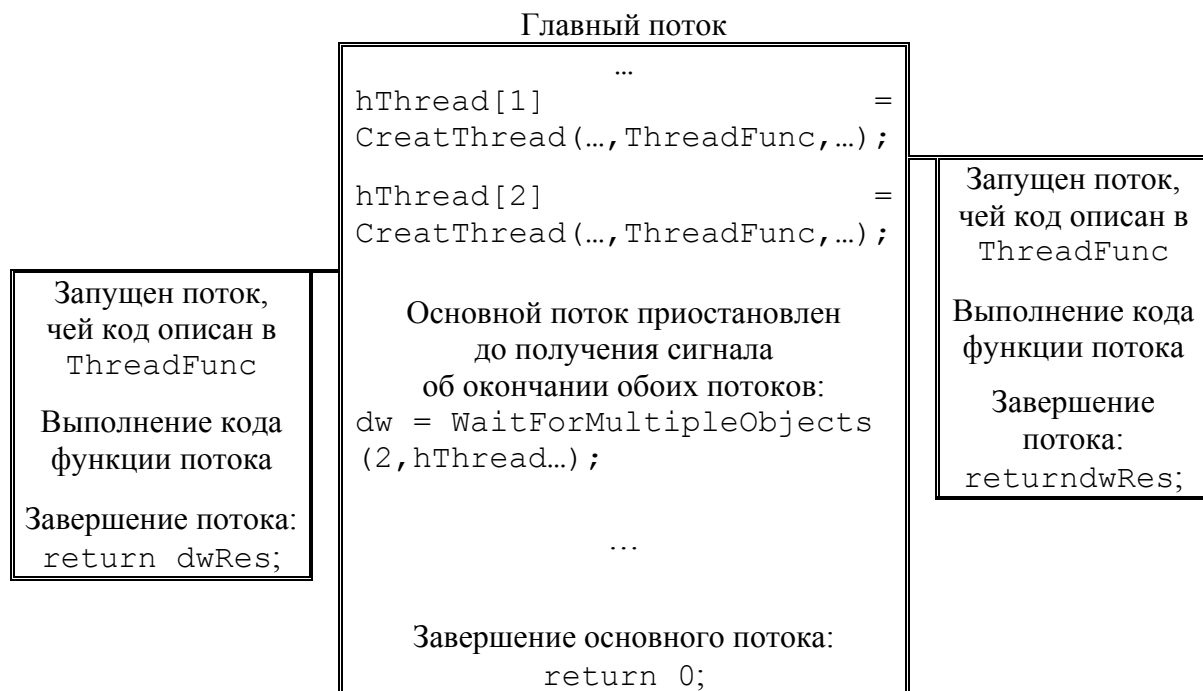


Рис. 2.2. Схема потоков примера

¹ Открепленным называют поток, для которого не существует других потоков в процессе, ожидающих его завершения.

При явном программировании многопоточности любая многопоточная программа должна состоять из основного потока и функций, в которых содержится код других потоков. Основной поток порождает хотя бы один дочерний поток, указывая также функцию, в которой находится код исполнения дочернего потока. Дочерние потоки при необходимости могут порождать свои дочерние потоки.

Пример 2.1 содержит код простого многопоточного приложения, использующего для создания потоков функции WinAPI. Подробно функции создания и управления потоками в Win32-средах рассматриваются в гл. 3. Заметим, что в этом примере основной поток не завершается до тех пор, пока не будут завершены все порожденные им потоки. Рис. 2.2 отображает схему потоков этого примера.

Пример 2.1. Простая многопоточная программа с использованием функций WinAPI

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

//Описание и инициализация глобальной константы,
//к ней имеют доступ все потоки и функции
const int n = 2;

//код функции потока
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    //DWORD WINAPI - тип возвращаемого потоком значения
    //ThreadFunc - имя функции потока (может быть произвольным)
    //PVOID - указатель на переменную произвольного типа
    //pvParam - используется для передачи данных в поток
    //Помещение переданных потоку данных (его номер)
    //в переменную потока num
    num = *((int *)pvParam);

    for (int i=0; i<100000; i++)
        fprintf(stdout, "Выполняется поток %d - %d\n", num, i);
    //Объявление и инициализация переменной,
    //содержащей значение, которое вернет поток
    DWORD dwRes = num;
    return dwRes; //возвращение управления в основной поток
}

//Код основного потока
int main(int argc, char** argv) {
    int x[n];
    //Объявление переменной, в которую функция создания потока
    //возвращает идентификатор нового потока
    DWORD dwThreadId[n];
    //Объявление переменной, в которую функция ожидания
    //завершения потоков вернет код завершения ожидания
```


Пример 2.1 (окончание). Простая многопоточная программа с использованием функций WinAPI

```

DWORD dw
//Объявление переменной, в которую будет считан
//результат выполнения потока
DWORD dwResult_main[n];
//Объявление массива описателей объектов ядра (потоков)
HANDLE hThread[n];
int i,j;
//Создание n потоков
for (int i=0; i<n; i++){
    x[i] = i;
    //Функция создания потока
    hThread[i]=CreateThread(NULL,0,ThreadFunc, (PVOID) &x[i],
                           0,&dwThreadId[i]);
    if(!hThread) printf("thread %d not execute!",i);
}
}

```

Существуют языки и библиотеки программирования, которые предполагают неявное порождение новых потоков, например, при выполнении некоторых операторов или с помощью директив компилятора. Один из таких подходов к организации многопоточности обсуждается в гл. 4.

2.2. Синхронизация потоков. Оператор ожидания

Далее будем предполагать следующее поведение вычислительной системы.

1. Значения базовых типов (например, `int`) хранятся в памяти и считываются или записываются неделимым образом. Другими словами, операция записи или чтения либо выполнена полностью (и между началом и окончанием этого выполнения ни одна другая операция не выполнялась), либо не выполнена вовсе (например, произошло обращение по несуществующему адресу), при этом система должна уметь обработать возникшую ошибку.

2. Значения обрабатываются в следующей цепочке неделимых операций: 1) если необходимо, значения считываются из памяти в регистры; 2) к ним применяются операции; 3) если необходимо, результаты записываются в память. Несмотря на то, что каждая из перечисленных операций неделима, вся цепочка таковой не является, т. е. прерывание работы потока может произойти между любыми из этих трех действий.

3. Каждый поток¹ имеет собственный набор регистров. Это реализуется либо предоставлением каждому потоку отдельного набора регистров, либо сохранением и восстановлением неделимым образом значений регистров при выполнении потока (смена контекста).

4. Промежуточные результаты потока сохраняются в регистрах или областях памяти, принадлежащих исполняемому потоку.

Рассмотрим простейшую многопоточную программу:

```
int y=0, z=0, x=0;
par x = y + z; \ y=1; z=2; endpar;
```

Потоки разделяют переменные x , y , z , выполняются параллельно и независимо, последовательность их выполнения не определена, более того, почти в любой момент поток может быть приостановлен, а затем продолжен. Предположим, что выражение $x = y + z$ реализуется загрузкой y в регистр с последующим прибавлением к нему z . В таком случае между загрузкой и прибавлением поток может быть прерван – и выполнится любое множество операций второго потока. Все это приводит к непредсказуемому поведению программы, а именно, значение переменной x может быть 0, 1 или 3.

Этот простой пример демонстрирует первую проблему синхронизации потоков при многопоточном программировании – *взаимное исключение*. Требуется отслеживать куски кода потока, которые должны выполняться неделимым образом. Соответствующий алгоритм принято называть *крупномодульным* решением задачи. При программировании следует реализовать механизм, который позволяет выполнять последовательность операций как одну неделимую. Такой код задает *мелкомодульное* решение задачи.

Второй проблемой синхронизации является *условное ожидание* (или синхронизация по условию). В этом случае поток приостанавливается до того момента, когда какое-то условие не будет выполнено. Поскольку поток бездействует, то выполнение условия возможно только в результате работы другого потока. Такие куски кода тоже следует отслеживать при крупномодульном решении и использовать какой-либо механизм реализации условного ожидания при программировании.

Определим общую нотацию синхронизации оператором [48]:

```
<await (B) S;>
```

¹ Поскольку в этой главе не обсуждается какая-либо конкретная реализация, понятия «процесс» и «поток» равноправны в том смысле, что они являются самостоятельными сущностями, участвующими в борьбе за разделяемые ресурсы (память, процессор и пр.). Далее везде в этой главе, не ограничивая общность, мы будем использовать термин «поток».

Булево выражение B задает условие задержки потока до момента, когда B станет истинным; список последовательных операторов S выполняется неделимым образом после достижения $B=TRUE$. При этом гарантируется, что вся последовательность операторов S будет выполнена неделимым образом при сохранении истинности условия B .

В примере 2.2 поток задерживается до возникновения условия положительности s , затем уменьшает s на 1, при этом гарантируется, что в момент уменьшения условие положительности s остается истинным.

Пример 2.2. Использование общей нотации синхронизации потоков

```
<await(s > 0) s = s-1;>
```

В рамках данного оператора взаимное исключение реализуется сокращенной записью:

```
<S;>
```

Условное ожидание реализуется также естественно:

```
<await(B) ;>
```

Оператор `await` является мощным и удобным в синхронизации. Однако его реализация в общей форме достаточно дорога. В дальнейшем мы будем использовать нотацию `await` при определении крупномодульных решений. В следующих параграфах рассмотрим возможную мелко модульную реализацию синхронизации для решения проблем исключительного доступа, барьерной и условной синхронизации потоков.

2.3. Взаимное исключение. Задача критической секции

В многопоточной ОС, как правило, одновременно выполняется несколько потоков, которые разделяют ресурсы компьютера. Не теряя общности, каждый поток можно считать последовательной программой.

Критической секцией (КС) называется последовательность операторов, имеющих доступ к разделяемому объекту.

Таким образом, КС – это фрагмент кода конкретного потока, обращающийся к ресурсу, который разделяется несколькими потоками. Говорят, что поток захватил КС, если он временно *монопольно* работает с разделяемым ресурсом. Говорят, что поток освободил КС, если он каким-либо образом уведомил другие потоки о прекращении своей работы с разделяемым ресурсом.

Предположим, что

- n потоков многократно выполняют сначала критическую, а потом не критическую секцию кода;

- любой поток, вошедший в КС, когда-нибудь ее покидает;
- поток завершается только вне КС.

Задача КС состоит в синхронизации потоков таким образом, чтобы в КС всегда находился только один поток. Впервые задачу критической секции была описал Э. Дейкстра в 1965 году.

С точки зрения программного кода решение задачи выражается в том, что КС всегда предшествует *протокол входа*, и за ней следует *протокол выхода*. Таким образом, предполагается следующее крупномодульное решение задачи КС:

```
thread CS (i=0; i<n) {
  while (true) {
    Протокол входа в КС;
    Критическая секция;
    Протокол выхода из КС;
    Некритическая секция;
  }
}
```

Протоколы входа и выхода должны удовлетворять следующим свойствам:

1. *Взаимное исключение*. В любой момент времени только один поток может выполнять критическую секцию.

2. *Отсутствие взаимной блокировки*. Если несколько потоков пытаются войти в критическую секцию и критическая секция свободна, то хотя бы один поток это осуществит.

3. *Отсутствие излишних задержек*. Если поток пытается войти в критическую секцию, а остальные потоки либо выполняют свои некритические секции, либо завершены, то первый поток немедленно войдет в критическую секцию.

4. *Возможность входа*. Поток, который пытается войти в критическую секцию, когда-нибудь это сделает.

Первые три свойства являются свойствами *безопасности*, четвертое – *живучести*. Реализация первых трех свойств является обязательной, дорогое четвертое свойство удовлетворяют не всегда.

Взаимодействие потоков в рамках задачи КС – пример синхронизации потоков взаимным исключением. Самыми низкоуровневыми механизмами реализации КС являются алгоритмы с активным ожиданием, использующие флаги.

Рассмотрим реализацию протоколов входа/выхода КС с помощью блокировок (флагов). Пусть `lock` – логическая переменная: `lock=true`, когда один из потоков находится в КС, и `lock=false` в противном случае. Пример 2.3 содержит крупномодульное решение задачи КС.

Пример 2.3. Крупномодульное решение задачи КС с помощью флагов

```

bool lock=false; //разделяемая переменная
thread CS (i=0; i<n) {
    while (true) {
        <await (!lock) lock=true;> //протокол входа
        Критическая секция;
        lock=false; //протокол выхода
        Некритическая секция;
    }
}

```

В угловых скобках указано действие, которое должно быть выполнено неделимым образом, т. е. на уровне инструкций процессора или каким-либо другим способом должна быть предоставлена возможность для проверки переменной и установки ее значения, а между этими действиями невозможно выполнение любой другой инструкции. Например, подходящей инструкцией для процессора является команда «проверить-установить» (test-set – TS). Результат действия инструкции описывает функция из примера 2.4.

Пример 2.4. Код инструкции «проверить-установить»

```

bool TS(bool *lock) {
    <bool initial=*lock; //сохранить начальное значение lock
    *lock=true;
    return initial;> //возвратить начальное значение lock
}

```

Реализация такой инструкции возможна практически на любом процессоре (например, инструкция инкремента с установкой кода условия, показывающего знак результата); TS() возвращает переданное ей значение аргумента без изменения. В то же время, в теле функции в true выставляется значение параметра, переданного по ссылке, т. е. после отработки функции значение аргумента всегда равно true вне зависимости от его первоначального значения. Этот факт можно использовать для реализации протокола входа в КС из примера 2.3. Пример 2.5 содержит *мелкомодульное решение задачи КС* на основе инструкции TS(). Отметим, что здесь и далее используется нотация Си для унарных операций: & – получение адреса и * – раскрытие ссылки.

Если критическая секция свободна (lock = false) и несколько потоков пытаются войти в нее, то первый из выполнивших инструкцию протокола входа TS(lock) скинет флаг КС на «занято» (lock = true), при

этом выйдет из цикла ожидания `while` протокола входа и получит возможность выполнить критический код. Все другие потоки останутся выполнять цикл `while` протокола входа до тех пор, пока поток, находящийся в КС, не выполнит протокол выхода, предоставив возможность входа в КС другому потоку. В примере 2.5 и далее оператор `skip` означает бездействие (пустое действие).

Пример 2.5. Мелкомодульное решение задачи КС с помощью флагов [35]

```
bool lock=false; //разделяемая переменная,
                //значение false означает «КС свободна»

thread CS (i=0;i<n) {
  while (true) {
    while (TS(&lock) ) skip; //протокол входа в КС
    Критическая секция;
    lock=false; //протокол выхода из КС
    Некритическая секция;
  }
}
```

Недостатком этого алгоритма является то, что `lock` все время перезаписывается, отсюда возникают конфликты при обращении к памяти. Не исключается и бесконечно долгое откладывание потока (другие потоки все время успешно входят в КС), т. е. алгоритм не обеспечивает выполнение условия живучести (4). Кроме того, для многопроцессорных архитектур с общей памятью возникает проблема когерентности кэшей.

Существует много решений задачи критической секции, обеспечивающих условие живучести (4) при слабом ограничении по стратегии планирования [48]. Рассмотрим для примера реализацию *алгоритма поликлиники* для n потоков, обеспечивающего вход в критическую секцию почти в порядке FIFO. Этот алгоритм был предложен Л. Лампортом [68].

Пусть `turn[n]` – массив целых чисел с начальными значениями 0. Протокол входа в КС следующий: поток `CS[i]` сначала присваивает переменной `turn[i]` значение, которое на 1 больше, чем максимальное среди текущих значений элементов массива `turn`. Затем поток ожидает, пока его значение `turn[i]` не станет наименьшим среди ненулевых значений массива `turn`. Выходя из КС, поток `CS[i]` присваивает `turn[i]` значение 0. В примере 2.6 содержится *крупномодульное решение алгоритма поликлиники* [48].

Для мелкомодульного решения необходимо реализовать два неделимых действия в протоколе входа. В алгоритме соответствующие строки помечены цифрами в комментариях.

Пример 2.6. Алгоритм поликлиники для задачи КС. Крупномодульное решение [48]

```
//Разделяемый массив очереди, инициализированный нулями
int turn[n]={[n] 0};

thread CS (i=0;i<n){
  while (true){
    //Начало протокола входа
    <turn[i] = max(turn)+1;> //см. пояснение 1
    for (j=0;j<n;j++)
      if (j!=i) //выполняется для всех j, за исключением j,
                //равного номеру процесса i
        <await(turn[j]==0 or turn[i]<turn[j]);>//см. пояснение 2
    //Конец протокола входа
    Критическая секция;
    turn[i]=0; //протокол выхода
    Некритическая секция;
  }
}
```

Пояснение 1. Требование неделимости операции поиска максимального элемента можно ослабить, если поток будет предупреждать о своем намерении войти в КС, присваивая своей переменной `turn[i]` значение 1. В результате несколько потоков могут получить одинаковые номера. Такие потоки можно упорядочить, например, по номеру потока `i`.

Пояснение 2. Второе неделимое действие можно реализовать циклом `while` с проверкой для потоков, ожидающих вход в КС, их значения в очереди. Поскольку несколько потоков могут иметь одинаковое значение очереди, то в расчет дополнительно принимается нумерация потоков в пуле (напомним, что каждый поток знает свой номер `i`). Если поток имеет номер очереди *не больше* всех остальных потоков и если его порядковый номер *наименьший* среди ожидающих потоков с таким же номером очереди, то он выполняется.

Пример 2.7. Алгоритм поликлиники для задачи КС. Мелкомодульное решение [48]

```
//Разделяемый массив очереди, инициализированный нулями
int turn[n]={[n] 0};
thread CS (i=0; i<n){
  while (true){
    //Начало протокола входа
    turn[i]=1; turn[i]=max(turn)+1; //см. пояснение 1
    for (j=0; j<n; j++)
      if (j!=i)
        while ((turn[j] != 0) and //см. пояснение 2
              ((turn[i]>turn[j]) or (turn[i]==turn[j] and i>j))) skip;
```

Пример 2.7 (окончание). Алгоритм поликлиники для задачи КС. Мелкомодульное решение [48]

```

//Конец протокола входа
    Критическая секция;
turn[i]=0; //протокол выхода
    Некритическая секция;
}
}

```

В результате *мелкомодульное решение задачи КС с помощью алгоритма поликлиники* можно записать кодом примера 2.7.

2.4. Сигнализирующие события. Барьерная синхронизация

Алгоритмы, параллельные по данным, часто требуют синхронизацию другого вида. Например, общим свойством итеративных алгоритмов последовательного приближения является зависимость результата последующей итерации от предыдущей. Если вычислением результата итерации занято несколько потоков, то для предотвращения проблемы синхронизации «гонка данных» необходимо, чтобы каждый из них не начинал следующей итерации, не дождавшись завершения предыдущей *всеми* потоками. *Крупномодульное* представление таких алгоритмов может быть следующим:

```

thread Woker (i=0; i<n){
    while (true){
        Код решения задачи i;
        Ожидание завершения всех n задач; //барьер
    }
}

```

Путей мелкомодульной реализации барьера существует много. Рассмотрим некоторые из них.

Барьер с разделяемым счетчиком

Заведем переменную-счетчик, которая подсчитывает количество потоков, завершивших итерацию (каждый поток, закончив работу, увеличивает счетчик на единицу). Тогда поток должен дожидаться заранее известного значения этой переменной.

Такой алгоритм требует неделимой операции увеличения счетчика на 1. Во многих процессорах есть команда «извлечь и сложить» (FA, Fetch

and Add). Основа кода мелко модульной реализации барьера с разделяемым счетчиком с учетом этой инструкции продемонстрирована в примере 2.8.

Пример 2.8. Барьер с разделяемым счетчиком

```
int count = 0;
thread Woker (i=0; i<n) {
    while (true) {
        код реализации задачи i;
        FA(count, 1); //<count = count + 1;>
        while (count != n) skip; //<await(count == n);>
    }
}
```

Однако данный код не полностью решает задачу барьера. Счетчик должен быть равен нулю в начале каждой итерации. Одно из возможных решений этой проблемы – использование двух счетчиков, один из которых увеличивается, а другой – уменьшается; они меняются ролями после каждой итерации (трудность в том, что смену ролей необходимо делать неделимым образом).

Кроме того, у решения примера 2.8 есть и другой недостаток – когда потоки ожидают, они постоянно читают один и тот же счетчик, этот же счетчик изменяют подошедшие к барьеру потоки. Для избавления от необходимости писать в одну и ту же переменную всеми потоками следует завести массив целочисленных счетчиков – по одному на каждый поток:

```
int arrive[n]=([n] 0);
```

Каждый поток «рапортует» о подходе к барьеру установкой своего флага `arrive[i]` в единицу, тогда сигналом того, что к барьеру подошли все потоки, является, например, выполнение условия

```
arrive[1]+arrive[2]+ ... + arrive[n] == n;
```

Однако проверка этого условия всеми потоками реализуется крайне неэффективно и опять приводит к конфликтам памяти.

Барьерная синхронизация с помощью флагов

Для корректного решения задачи барьерной синхронизации возможно, например, 1) использовать два целочисленных массива

```
int arrive[n]=([n] 0), continue[n]=([n] 0);
```

сигнализирующих о прохождении барьера всеми потоками, а также 2) создать отдельный поток `Coordinator` для управления массивом `continue`.

Рабочие потоки, выполнив свою часть итерации и подойдя к барьеру, сигнализируют об этом через переменную `arrive[i]`, после чего приостанавливаются.

навливаются и ожидают сигнала от потока Coordinator через переменную `continue[i]`. Управляющий поток сначала ожидает, пока все `arrive[i]` не станут равными 1 (все потоки подойдут к барьеру), затем сигнализирует об этом, изменяя массив `continue`.

Крупномодульное решение барьерной синхронизации с помощью флагов с управляющим потоком [48] демонстрируется в примере 2.9. Неделимый оператор ожидания `await` в данном случае легко реализуется оператором `while`.

Пример 2.9. Крупномодульное решение барьерной синхронизации с помощью флагов

<pre>int arrive[n]=([n] 0), continue[n]=([n] 0); thread Woker (i=0; i<n){ while (true) { Код решения задачи i; arrive[i] = 1; //Возможна реализация //с помощью цикла while //(укажите, как) <await(continue[i] == 1);> continue[i] = 0; } }</pre>	<pre>thread Coordinator{ while (true){ for (i=0; i<n; i++){ //Возможна реализация //с помощью цикла while //(укажите, как) <await (arrive[i]==1);> arrive[i] = 0; } for (i=0; i<n; i++) continue[i] = 1; } }</pre>
---	--

Отметим преимущества решения примера 2.9: 1) каждый рабочий поток записывает в свою переменную, а управляющий поток только читает их; 2) каждый рабочий поток проверяет (читает) свою переменную, а управляющий поток записывает в них.

Элементы массивов `arrive` и `continue` называются *флагами*. Алгоритм должен предусматривать не только корректную установку флагов, но и их сброс после прохождения барьера. Таким образом, для предотвращения гонок для синхронизации барьером необходимы два набора флагов.

Правило синхронизации с помощью флагов:

1) флаг синхронизации сбрасывается только потоком, ожидающим его установки (это гарантирует, что флаг не будет сброшен до того, как обратили внимание на то, что он выставлен);

2) флаг нельзя устанавливать до тех пор, пока не известно, что он сброшен.

Алгоритмы, рассмотренные выше, называют синхронизацией с активным ожиданием. Их реализация эффективна, если каждый поток выполняется на своем процессоре, поэтому к недостаткам такого подхода

можно отнести наличие холостого с точки зрения итерационного алгоритма потока Coordinator. Более того, этот поток, проверяя установку флагов `arrive[i]` в цикле по очереди, будет тормозить рабочие потоки, которые подойдут к барьеру почти одновременно (так обычно и бывает в согласованном параллельном программировании).

Оба недостатка можно преодолеть, объединив функции рабочего и управляющего потоков так, чтобы каждый рабочий поток был одновременно и управляющим. Для этого можно организовать рабочие потоки в *дерево* (по принципу дихотомии), тогда потоки по организации барьера поделятся на три группы – корневой, промежуточный узлы и узел-лист.

Существуют алгоритмы реализации *симметричных* барьеров (все потоки выполняют один код). В таких алгоритмах каждый поток играет роли и рабочего, и управляющего. Синхронизация происходит в несколько этапов, на каждом из них синхронизируются два потока (при подходе к барьеру поток сигнализирует установкой своего флага и ожидает установку флага другого потока, затем сбрасывая его). Схема выбора потока, с которым следует синхронизироваться на очередном уровне, строится таким образом, чтобы на последнем этапе произошла неявная синхронизация всех потоков.

Если количество рабочих потоков кратно степени двойки, то их можно синхронизировать *симметричным барьером-бабочкой* (рис. 2.3), который состоит из $\log_2 n$ уровней (n – количество потоков). На уровне s синхронизируются потоки, находящиеся на расстоянии 2^{s-1} друг от друга.

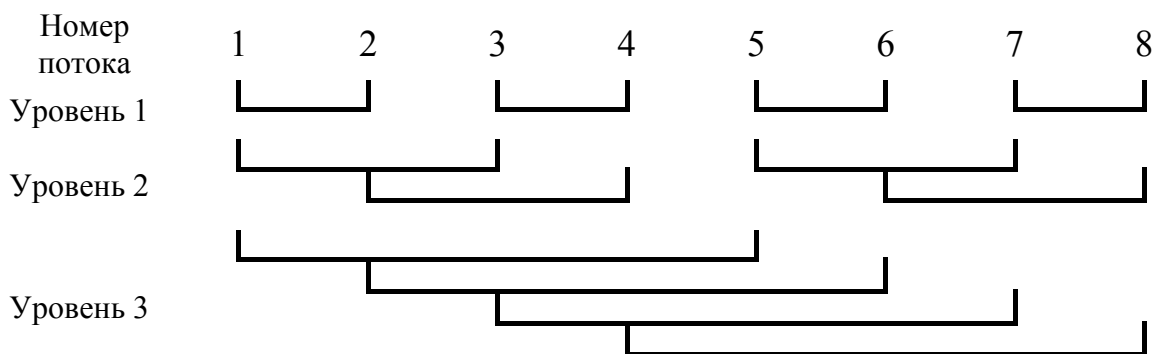


Рис. 2.3. Барьер-бабочка для восьми потоков

Если количество потоков не кратно степени двойки, то можно провести синхронизацию *барьером с распространением* (рис. 2.4, 2.5), в котором также на каждом s -м уровне синхронизируются процессы на расстоянии 2^{s-1} друг от друга, но последовательность синхронизации несколько иная. Как видно из рис. 2.4 описанная схема синхронизации дает гарантию барьера «с запасом» – на последнем уровне некоторые процессы успевают неявно

синхронизироваться дважды. Например, процесс 3 синхронизируется с процессом 4 один раз явно (на первом уровне), а во второй раз опосредованно, проходя на третьем уровне синхронизацию с процессом 1, который, в свою очередь, неявно синхронизировался с процессом 4 на втором уровне барьера.

уровень синхронизации s ; шаг синхронизации $j = 2^{s-1}$ на уровне s																									
$s = 1; j = 1$								$s = 2; j = 2$								$s = 3; j = 4$									
0	ожидает прохождения уровня процессом 1:								ожидает прохождения уровня процессом 2:								ожидает прохождения уровня процессом 4:								
	0	1	2	3	4	5			0	1	2	3	4	5			0	1	2	3	4	5			
	в конце уровня процесс знает о прохождении барьера процессами:																								
			0	1							0	1	2	3					0	1	2	3	4	5	0
1	ожидает прохождения уровня процессом 2:								ожидает прохождения уровня процессом 3:								ожидает прохождения уровня процессом 5:								
	0	1	2	3	4	5			0	1	2	3	4	5			0	1	2	3	4	5			
	в конце уровня процесс знает о прохождении барьера процессами:																								
			1	2							1	2	3	4					1	2	3	4	5	0	1
2	ожидает прохождения уровня процессом 3:								ожидает прохождения уровня процессом 4:								ожидает прохождения уровня процессом 0:								
	0	1	2	3	4	5			0	1	2	3	4	5			0	1	2	3	4	5			
	в конце уровня процесс знает о прохождении барьера процессами:																								
			2	3							2	3	4					2	3	4	5	0	1	2	3
3	ожидает прохождения уровня процессом 4:								ожидает прохождения уровня процессом 5:								ожидает прохождения уровня процессом 1:								
	0	1	2	3	4	5			0	1	2	3	4	5			0	1	2	3	4	5			
	в конце уровня процесс знает о прохождении барьера процессами:																								
			3	4							3	4	5	0				3	4	5	0	1	2	3	4
4	ожидает прохождения уровня процессом 5:								ожидает прохождения уровня процессом 0:								ожидает прохождения уровня процессом 2:								
	0	1	2	3	4	5			0	1	2	3	4	5			0	1	2	3	4	5			
	в конце уровня процесс знает о прохождении барьера процессами:																								
			4	5							4	5	0	1				4	5	0	1	2	3	4	5
5	ожидает прохождения уровня процессом 0:								ожидает прохождения уровня процессом 1:								ожидает прохождения уровня процессом 3:								
	0	1	2	3	4	5			0	1	2	3	4	5			0	1	2	3	4	5			
	в конце уровня процесс знает о прохождении барьера процессами:																								
			5	0							5	0	1	2				5	0	1	2	3	4	5	0

Рис. 2.4. Барьер с распространением для 6 процессов: прохождение барьера каждым процессом с пояснением, какие процессы с точки зрения каждого из них гарантированно прошли барьер по окончании каждого уровня

Пример 2.10 содержит основу кода крупномодульного решения симметричной барьерной синхронизации с помощью флагов [48].

На уровне s синхронизация происходит следующим образом. Поток с номером i , подойдя к барьеру, убеждается в том, что его флаг сброшен, это означает успешное прохождение барьера потоком, с которым поток i синхронизировался на предыдущем уровне $s-1$. Далее поток устанавливает свой флаг прибытия к барьеру, определяет номер потока, с которым следует синхронизироваться на текущем уровне прохождения барьера, ожидает сообщения о прибытии этого потока к барьеру и затем сбрасывает его флаг.

Приведенный алгоритм не совсем верен. Если потоки работают с разной скоростью, то велика вероятность того, что синхронизируются потоки, проходящие разные уровни s барьера, иными словами, не исключено состояние гонок.

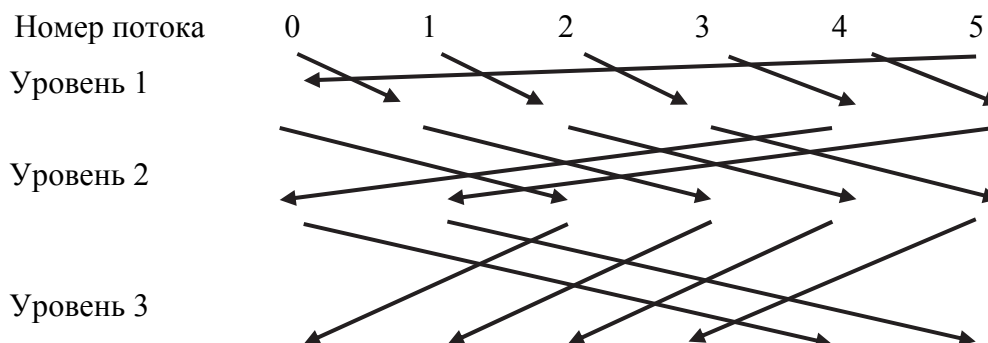


Рис. 2.5. Барьер с распространением для шести потоков

Для исправления ситуации можно, например, на каждом уровне синхронизации использовать свои флаги, т. е. использовать двумерный массив флагов `arrive[n][num_level]`. Другим решением является использование флагов, которые могут принимать значения больше единицы (пример 2.11). Тогда вместо строк (1)–(2) поток, приходя на новый уровень, увеличивает свой флаг на единицу:

```
arrive[i] = arrive[i] + 1;
```

Затем ожидает прохождения барьера потоком j как минимум на том же уровне синхронизации. В результате код (3)–(4) следует заменить оператором ожидания следующего вида:

```
<await (arrive[j] >= arrive[i]);>
```

который в данном случае можно реализовать циклом¹:

```
while (arrive[j] < arrive[i]) skip;
```

¹ Вообще для реализации нотации условной синхронизации `<await B>` чаще всего можно использовать цикл `while (!B) skip`.

Пример 2.10. Основа решения для реализации симметричного барьера

```

int arrive[n]=([n] 0);
//Количество уровней синхронизации,
//определяется количеством потоков и схемой синхронизации
int num_level = ...
thread Woker (i=0; i<n){
    int s; //счетчик уровней прохождения барьера
    while (true){
        Код решения задачи i;
        //Код барьера
        for (s=1; s<=num_level; s++){
            //Проверка: сброшен ли свой флаг, установленный
            //для другого потока на предыдущей итерации
(1)  <await (arrive[i] == 0);>
(2)  arrive[i] = 1; //установка своего флага
            Определить номер потока j для синхронизации на уровне s
            //Ожидание прибытия к барьеру j-го потока,
            //который выставит флаг для i-го потока
(3)  <await (arrive[j] == 1);>
(4)  arrive[j] = 0; //сброс флага j-го потока
        }
    }
}

```

Пример 2.11. Крупномодульное решение симметричной барьерной синхронизации

```

int arrive[n]=([n] 0);
//Количество уровней синхронизации,
//определяется количеством потоков и схемой синхронизации
int num_level = ...
thread Woker (i=0; i<n){
    int s; //счетчик уровней прохождения барьера
    while (true){
        Код решения задачи i;
        //Код барьера
        for (s=1; s<=num_level; s++){
            //Установка своего флага на очередном уровне синхронизации
            arrive[i] ++;
            Определить номер потока j для синхронизации на уровне s
            //Ожидание прибытия к барьеру j-го потока,
            //который выставит флаг для i-го потока
            //на уровне синхронизации не ниже s
            <await (arrive[j]>= arrive[i]);>
        }
    }
}

```

Если итераций неограниченное число, есть опасность переполнения флагов, поскольку теперь их никто не сбрасывает. Однако на практике вероятность переполнения невелика.

Любая программная реализация барьеров приводит к большим накладным расходам. Существует и аппаратная поддержка барьерной синхронизации, которая позволяет свести эти расходы к минимуму.

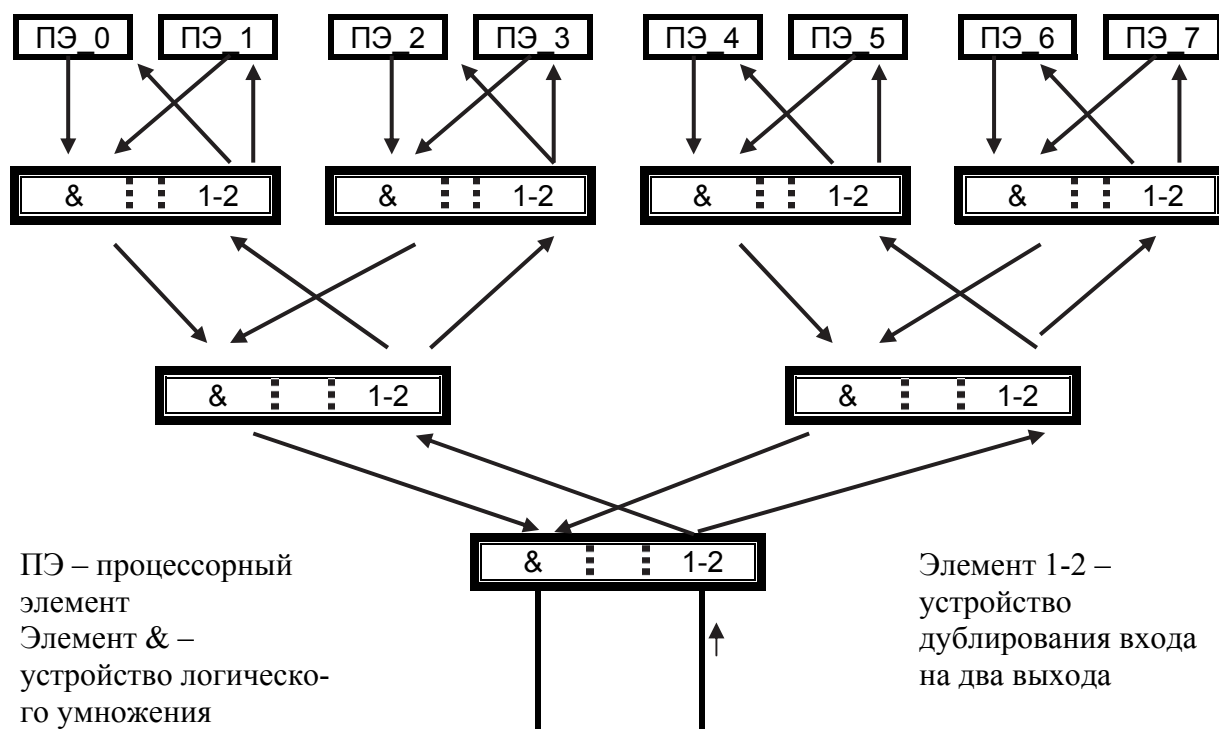


Рис. 2.6. Барьерная синхронизация в компьютере Cray T3D/T3E [9]

Например, в вычислительной системе с распределенной памятью Cray T3D/T3E разработана аппаратная барьерная синхронизация по принципу дихотомии (рис. 2.6) [9]. В схеме поддержки каждого процессорного элемента (ПЭ) предусмотрено несколько входных и выходных регистров синхронизации. Каждый разряд этих регистров соединен со своей независимой цепью реализации барьера. Все цепи синхронизации одинаковы, а их общее число зависит от конфигурации компьютера. Каждая цепь строится по принципу двоичного дерева на основе двух типов устройств. Одни устройства реализуют логическое умножение (&), другие дублируют сигнал с единственного входа на два выхода.

2.5. Семафоры

Протоколы входа в КС и барьерной синхронизации, рассмотренные выше, называют протоколами с активным ожиданием (поток не просто ожидает, а все время проверяет переменную для синхронизации). Такие алгоритмы имеют ряд недостатков: они весьма сложны, кроме того, нет четкой грани между переменными для синхронизации и переменными для вычислений, узким местом становится доступ к переменной, разделяемой всеми потоками, которую все время считывают (проверяют) и часто изменяют. Одним из специальных средств синхронизации являются семафоры, которые впервые предложил Эдсгер Дейкстра для реализации взаимного исключения в 1968 году.

Семафор – это особый тип разделяемой переменной, которая обрабатывается только двумя *неделимыми* операциями $P()$ и $V()$. Семафор можно считать экземпляром класса «семафор», операции $P()$ и $V()$ – методами этого класса с дополнительным атрибутом, определяющим их неделимость. Значение семафора – *неотрицательное целое число*.

Операция $V()$ используется для сигнализации того, что событие произошло, поэтому она неделимо увеличивает значение семафора на единицу.

Операция $P()$ приостанавливает поток до момента, когда произойдет некоторое событие, поэтому она, дождавшись, когда значение семафора станет положительным, уменьшает его значение на единицу. Выполнение операции $P()$ не может быть приостановлено, т. е. гарантируется, что непосредственно перед уменьшением значение семафора положительно.

В нашей нотации будем объявлять семафоры следующим образом:

```
sem s;
```

По умолчанию значение семафора 0, но он может быть проинициализирован любым положительным значением:

```
sem s=1;
```

Возможны объявление и инициализация массива семафоров:

```
sem sarray[10] = ( [10] 0 );
```

После инициализации семафор можно обрабатывать только операциями $P()$ и $V()$. Каждая из них является неделимым действием с семафором в качестве единственного аргумента:

```
P(s): <await (s>0) s=s-1;>
```

```
V(s): <s=s+1;>
```

Операция $P()$ гарантирует неотрицательность семафора.

Пусть $s=1$. Если два потока одновременно пытаются выполнить операцию $P(s)$, то это удастся только одному из них. Если один поток пыта-

ется выполнить операцию $V(s)$, а другой – $P(s)$, то будут выполнены обе операции, но в непредсказуемом порядке, причем в результате значение s не изменится.

Обычный семафор может принимать любые неотрицательные целые значения, двоичный семафор – только значения 0 и 1. Следовательно, операция $V()$ для двоичного семафора будет выполнена, только если его значение 0:

```
V(sbin): <await (sbin=0) sbin=sbin+1;>
```

Семафоры могут быть реализованы как аппаратно, так и программно.

Решение задачи критической секции с использованием семафоров

Семафоры были придуманы, прежде всего, как механизм решения задачи КС, которая в их терминах записывается весьма естественно (пример 2.12). Действительно, пусть с критической секцией связан двоичный семафор `mutex` с *единичным начальным значением*, соответствующим свободному состоянию КС. Тогда значение `mutex=0` будет означать, что КС занята. Операция $P(mutex)$ суть реализация для флага `bool lock` протокола входа `<await (!lock) lock=true;>` из общего решения задачи КС примера 2.3. В свою очередь, операция $V(mutex)$ реализует протокол выхода `<lock=false;>` из КС.

Обеспечение безопасного доступа к пересекающемуся множеству ресурсов с использованием семафоров

Классическая задача об обедающих философах [16] демонстрирует использование семафоров для обеспечения безопасного доступа к пересекающемуся множеству ресурсов.

Пример 2.12. Решение задачи критической секции с помощью семафоров

```
// семафор: mutex=1 => КС свободна, mutex=0 => КС занята
sem mutex=1;
thread CS (i=0;i<n){
  while (true){
    P(mutex); //протокол входа
    Критическая секция;
    V(mutex); // протокол выхода
    Некритическая секция;
  }
}
```

Задача об обедающих философах. Пять философов сидят возле круглого стола. Они проводят жизнь, чередуя приемы пищи и размышления. В центре стола находится большое блюдо спагетти. Спагетти длинные и запутанные, философам тяжело управляться с ними, поэтому каждый из них, чтобы съесть порцию, должен пользоваться двумя вилами. К несчастью, философам дали *всего пять* вилок. Между каждой парой философов лежит одна вилка, поэтому эти высококультурные и предельно вежливые люди договорились, что каждый будет пользоваться только теми вилами, которые лежат рядом с ним (слева и справа).

Требуется написать программу, моделирующую поведение философов. Программа должна избегать фатальной ситуации, в которой все философы голодны, но ни один из них не может взять обе вилки (например, каждый из философов держит по одной вилке и не хочет отдавать ее).

При решении задачи вилки можно представить массивом семафоров `sem fork[5]` (состоянию «*i*-я вилка свободна» соответствует `fork[i]=1`). Для безопасного пользования вилами процесс еды должен быть оформлен критической секцией (философы в нашем алгоритме, как многие выдающиеся люди, левши):

```
//взять левую вилку, затем правую
P(fork[i]); P(fork[(i+1)%5]);
  Поесть;
//положить правую вилку, затем левую
V(fork[(i+1)%5]); V(fork[i]);
```

Поскольку процесс взятия обеих вилок одним философом не является неделимой операцией, то при выполнении всеми философами такого симметричного кода может возникнуть взаимная блокировка. Тупиковая ситуация будет спровоцирована, если каждый философ возьмет по одной вилке (в данном случае левой) и будет бесконечно удерживать их, не имея возможности взять правую вилку, поскольку она заблокирована соседом. Одним из возможных решений без взаимной блокировки является появление потока-«правши», который берет сначала правую вилку, а потом левую (пример 2.13).

Возможна реализация и симметричного решения задачи о философах. Для этого достаточно сделать процесс захвата правой и левой вилок неделимой операцией (пример 2.14).

Отметим, что механизмом семафора не обеспечивается выполнение свойства живучести для протокола входа. Если реализация не предполагает обратного, то никакой очереди к семафору не устанавливается, и в общем случае может существовать процесс, навечно задержавшийся на операции `P()`.

Пример 2.13. Решение задачи об обедающих философах

```
sem fork[5] = {1, 1, 1, 1, 1};
thread Philosopher(i=0;i<5){
  while (true){
    if (i<4){
      //Взять левую вилку, затем правую
      P(fork[i]); P(fork[(i+1)%5]);
      Поесть;
      //Положить правую вилку, затем левую
      V(fork[(i+1)%5]); V(fork[i]);
      Поразмыслить;
    }
    else { //код философа-правши
      //Взять правую вилку, затем левую
      P(fork[0]); P(fork[4]);
      Поесть;
      //Положить правую вилку, затем левую
      V(fork[4]); V(fork[0]);
      Поразмыслить;
    }
  }
}
```

Пример 2.14. Симметричное решение задачи об обедающих философах

```
//Семафор для получения исключительных прав
//на часть разделяемого ресурса
sem fork[5] = {1,1,1,1,1};
//Семафор для выполнения операции захвата
//разделяемого ресурса по частям как неделимой
sem cs=1;
thread Philosopher(i=0;i<5){
  while (true){
    P(cs);
    //Начало захвата разделяемого ресурса по частям
    //Взять левую вилку, затем правую
    P(fork[i]); P(fork[(i+1)%5]);
    V(cs); // ресурс захвачен
    Поесть;
    //положить правую вилку, затем левую
    V(fork[(i+1)%5]); V(fork[i]);
    Поразмыслить;
  }
}
```

Реализация барьеров с использованием семафоров

Рассмотрим реализацию барьерной синхронизации с управляющим потоком с помощью семафоров (пример 2.15). Как и в случае реализации барьера с помощью флагов, алгоритм должен обеспечивать: 1) многократное использование барьера; 2) задержку потока на барьере до тех пор, пока к нему не подошли все потоки.

При реализации барьеров семафор используется в качестве флага синхронизации, такие семафоры называются *сигнализирующими*. Сигнализирующий семафор, как правило, имеет *нулевое начальное значение* (в отличие от семафоров, использующихся в протоколах КС).

Пример 2.15. Реализация барьера с управляющим потоком

<pre>sem arrive=0, continue[1:n]=([n] 0); thread Woker (i=1;i<n){ while (true){ Код решения задачи i; V(arrive); P(continue[i]); } }</pre>	<pre>thread Coordinator{ while (true){ for (i=1;i<n;i++) P(arrive); for (i=1;i<n;i++) V(continue[i]); } }</pre>
---	---

Поскольку действие операции $V()$ для семафора «запоминается», то для реализации алгоритма требуется меньше семафоров, чем переменных флагов (сравните коды примеров 2.9 и 2.15). Для реализации барьера необходим общий семафор `arrive` и собственный для каждого рабочего потока семафор `continue[i]`. Каждый поток сигнализирует о своем прибытии к барьеру, выполняя операцию $V(arrive)$ для общего семафора, а затем ожидает прибытия к барьеру всех оставшихся потоков, выполняя для своего второго семафора операцию $P(continue[i])$. Управляющий поток, в свою очередь, сначала ожидает прибытия всех потоков к барьеру, в цикле выполняя операцию $P(arrive)$, затем сигнализирует рабочим потокам о возможности продолжения, выполняя для их семафоров операции $P(continue[i])$.

Легко также построить симметричный барьер с использованием семафоров. Во многих реализациях параллельных языков и библиотек существует неделимая операция `barrier`, выполняющая барьерную синхронизацию.

Условная синхронизация с помощью семафоров

С помощью семафоров можно программировать условное ожидание. Продемонстрируем это на примере решения задачи о кольцевом буфере, классической для парадигмы «производитель – потребитель».

Задача о кольцевом буфере. Потоки «производитель» и «потребитель» разделяют буфер, состоящий из n ячеек. Производитель передает сообщение потребителю, помещая его в конец очереди буфера. Потребитель сообщение извлекает из начала очереди буфера.

Требуется организовать взаимодействие потоков таким образом, чтобы избежать состояния гонок данных: 1) производитель не может положить сообщение в полный буфер; 2) потребитель не может забрать сообщение из пустого буфера; 3) два производителя не должны записывать сообщение в одну и ту же ячейку, если оно между этими двумя операциями записи не была прочитано хотя бы одним потребителем; 4) два потребителя не должны прочесть одно и то же сообщение.

Рассмотрим сначала решение для случая одного производителя и одного потребителя. Представим буфер массивом `buf[n]`, $n > 1$ (можно использовать связный список). Пусть `front` – индекс первого сообщения в очереди; `rear` – индекс первой пустой ячейки после сообщения в конце очереди. Тогда, выполняя последовательность операторов

```
buf[rear] = data; rear = (rear+1) % n;
```

производитель помещает сообщение в буфер, а выполняя последовательность операторов

```
result = buf[front]; front = (front+1) % n;
```

потребитель извлекает сообщение из буфера.

В этом случае синхронизация по условию нужна для того, чтобы сообщение нельзя было забрать из пустой очереди и поместить в полный буфер. Семафоры используются аналогично флагам.

Для обеспечения безопасной записи сообщения в ячейку буфера требуется n -местный семафор `empty` с начальным значением n . Поток-потребитель, забирая значение из буфера, каждый раз увеличивает значение семафора `empty` на единицу, выполняя над ним операцию $V(empty)$, сигнализируя, что произошло событие «опустошить буфер». Поток-производитель перед тем, как положить сообщение в буфер, выполняет для семафора `empty` операцию $P(empty)$, убеждаясь в наличии в буфере свободных ячеек.

Для обеспечения безопасного изъятия данных из буфера требуется второй n -местный семафор `full` с начальным значением 0. Для него поток-производитель выполняет операцию $V(full)$ при каждом пополнении буфера, а процесс-потребитель операцией $P(full)$ убеждается в возможности безопасного извлечения данных.

Реализация кольцевого буфера с использованием семафоров для двух процессов [48] продемонстрирована в примере 2.16.

Пример 2.16. Решение задачи о кольцевом буфере для двух процессов

```

typeT buf[n]; //массив некоторого типа typeT
int front = 0, rear = 0;
sem empty = n, full = 0;
thread Producer{
    while (true){
        Создать сообщение data;
        //Критическая секция: поместить data в буфер
        P(empty);
        buf[rear] = data; rear = (rear+1) % n;
        V(full);
    }
}
thread Consumer{
    while (true){
        //Критическая секция: извлечь сообщение из буфера
        P(full);
        result = buf[front]; front = (front+1) % n;
        V(empty);
        Потребить result
    }
}

```

Пример 2.17. Реализация задачи о кольцевом буфере для нескольких процессов-производителей и нескольких процессов-потребителей с помощью семафоров

```

typeT buf[n]; //массив некоторого типа typeT
int front = 0, rear = 0;
sem empty = n, full = 0; //для условной синхронизации
sem mutexP = 1, mutexC = 1; //для взаимного исключения

thread Producer(i=1;i<P){
    while (true){
        Создать сообщение data;
        //Поместить data в буфер
        P(empty);
        P(mutexP);
        buf[rear] = data;
        rear = (rear+1) % n;
        V(mutexP);
        V(full);
    }
}

thread Consumer(j=1;j<C){
    while (true){
        //Извлечь сообщение из буфера
        P(full);
        P(mutexC);
        result = buf[front];
        front = (front+1) % n;
        V(mutexC);
        V(empty);
        Потребить result
    }
}

```

В случае наличия нескольких производителей и нескольких потребителей необходимо исключать ситуацию, когда несколько процессов помещают сообщение в одну ячейку буфера или извлекают сообщение из одной и той же ячейки (пример 2.17). Для этого следует завести еще два семафора-

ра для обеспечения критических секций – отдельно для производителя, отдельно для потребителя. В нашем примере это семафоры `mutexP` («сейчас что-то положу, и мне не мешать») и `mutexC` («сейчас что-то извлеку, и мне не мешать»). Получается, что пара семафоров `empty` и `full` используется для условной синхронизации, а пара семафоров `mutexP` и `mutexC` – для взаимного исключения.

Комбинация параллельного и исключительного доступа к разделяемому ресурсу

Часто возникает ситуация, когда к одному и тому же ресурсу необходимо реализовать и совместный, и исключительный доступ. Рассмотрим решение классической задачи о читателях и писателях.

Задача о читателях и писателях. Базу данных разделяют два типа процессов – читатели и писатели. Читатели выполняют транзакции, которые просматривают записи базы данных, а транзакции писателей и просматривают, и изменяют записи. Предполагается, что вначале БД находится в непротиворечивом состоянии (т. е. отношения между данными имеют смысл).

Требуется реализовать взаимодействие процессов-писателей и процессов-читателей с базой данных и между собой следующим образом. Каждая отдельная транзакция (чтение или запись) переводит БД из одного непротиворечивого состояния в другое. Для предотвращения взаимного влияния транзакций процесс-писатель должен иметь исключительный доступ к БД. Если к БД не обращается ни один из процессов-писателей, то выполнять транзакции могут одновременно сколько угодно читателей.

Пусть `num_r` – количество процессов-читателей, а `num_w` – количество процессов-писателей, получивших доступ к БД. Для правильной сериализации транзакций необходимо, чтобы в любой момент времени соотношение между количеством писателей и читателей, получивших доступ к БД, обеспечивало истинность следующего логического выражения: $R-W := ((num_r == 0 \text{ or } num_w == 0) \text{ and } num_w \leq 1)$. Крупномодульное решение задачи о читателях и писателях [48], сохраняющее истинность $R-W$, приведено в примере 2.18.

Однако решение из примера 2.18 не определяет порядок, в котором процессы получают доступ к БД. Кроме того, условия защиты операторов `await` в протоколах входа в КС для обновления счетчиков `num_r` и `num_w` в процессах писателей (`num_r == 0 and num_w == 0`) и читателей (`num_w == 0`) частично перекрываются, что затрудняет реализацию этих неделимых действий даже с помощью семафоров.

Рассмотрим решение, использующее *метод передачи эстафеты* [48], который может применяться для одновременной реализации нескольких операторов ожидания самого общего вида `<await(B_i) S_i;>`.

Пример 2.18. Крупномодульное решение задачи о читателях и писателях с помощью семафоров

<pre> int num_r = 0, num_w = 0; thread Reader(i=0; i<R) { while (true) { ... //(1) <await(num_w == 0) num_r = num_r + 1;> Читать базу данных; <num_r = num_r - 1;> } } </pre>	<pre> thread Writer(j=0; j<W) { while (true) { ... //(2) <await(numr == 0 and num_w == 0) num_w = num_w + 1;> Записать в базу данных; <num_w = num_w - 1;> } } </pre>
--	--

Метод эстафеты реализует *разделенный двоичный семафор* `sem_split`, представленный 1) одним двоичным семафором `sem_cs` с начальным значением 1 для управления входом в любое неделимое действие, связанное с распределенным семафором; 2) одним семафором `delay_sem_i` и одним счетчиком `delay_count_i` с нулевыми начальными значениями, связанными с каждым условием защиты `B_i` *i*-го оператора `await`. Семафор `delay_sem_i` используется для приостановки процесса до момента, когда условие защиты станет истинным, а в счетчике `delay_count` хранится число приостановленных процессов.

Каждый оператор ожидания `<await(B_i) S_i;>` рассматривается как часть обобщенной критической секции для последовательности операторов `<S_i;>`. Пример 2.19 демонстрирует реализацию протокола входа в такую критическую секцию для различных форм записи оператора ожидания. Этот код можно считать обобщенной операцией `P(sem_split_i)` для разделенного двоичного семафора `sem_split_i`, поскольку в результате выполняется операция `P()` только для одного из семафоров `sem_split_i`, связанных с условиями защиты операторов `await`. После этого гарантированно неделимо и со взаимным исключением может быть выполнена последовательность операторов `<S_i;>`. Отметим, что замена оператора `await` оператором `if` корректна только при соответствующем коде протокола выхода из КС для распределенного семафора, который и обеспечивает передачу эстафеты.

Протокол выхода из неделимых действий в случае метода передачи эстафеты программируется таким образом, чтобы сигнал получил только один из семафоров, входящих в состав разделенного двоичного семафора `sem_split`, реализуя его операцию `V(sem_split_i)`. При этом протокол

выхода в методе передачи эстафеты можно использовать для точного управления порядком доступа к разделяемому ресурсу.

Пример 2.19. Код протоколов входа в обобщенную критическую секцию в зависимости от вида оператора ожидания

<pre>//<await(B_i) S_i;> P(cs); if (not B_i){ delay_count_i = delay_count_i + 1; V(cs); P(delay_sem_i); } S_i;</pre>	<pre>//<S_i;> P(cs); S_i;</pre>
--	---------------------------------------

Проиллюстрируем сказанное, применив метод передачи эстафеты для реализации крупномодульного решения задачи о читателях и писателях из примера 2.18. Рассмотрим разделенный двоичный семафор, реализующий два оператора `await` (в коде примера 2.18 помечены (1), (2)). Наш распределенный семафор должен состоять из следующих компонент:

- `sem cs=1;` – семафор, используемый для управления входом в любое неделимое действие;
- `sem rd=0;` – семафор, связанный с условием защиты в процессе-читателе. Операция `P(rd)` приостанавливает процесс-читатель. Операция `V(rd)` запускает один из приостановленных процессов-читателей;
- `int delay_rd=0;` – счетчик приостановленных процессов-читателей;
- `sem wrt=0;` – семафор, связанный с условием защиты в процессе-писателе. Операция `P(wrt)` приостанавливает процесс-писатель. Операция `V(wrt)` запускает один из приостановленных процессов-писателей;
- `int delay_wrt=0;` – счетчик приостановленных процессов-писателей.

При кодировании протокола выхода реализуем, например, преимущество читателей, т. е. в нашем случае порядок получения сигнала в разделенном двоичном семафоре должен быть следующим:

- 1) если нет активных писателей, но есть приостановленный читатель, то он продолжается;
- 2) если нет активных читателей или писателей, но есть приостановленный писатель, то он продолжается;
- 3) если нет отложенных процессов, которые можно безопасно продолжать, то семафор `cs` получает сигнал.

Таким образом, общая для всех часть протокола выхода реализуется кодом `V_sem_split` примера 2.20.

Пример 2.20. Код общей части протокола выхода V_sem_split

```

if (num_w == 0 and delay_rd > 0){
    //Возобновить процесс-читатель, или
    delay_rd = delay_rd-1; V(rd);
}
//возобновить процесс-писатель, или
else if (num_r == 0 and num_w == 0 and delay_wrt > 0){
    delay_wrt = delay_wrt-1; V(wrt);
}
else V(cs); //освободить блокировку входа

```

Пример 2.21 содержит код решения задачи о читателях и писателях методом передачи эстафеты, в котором протоколы выхода заменены ссылкой на общий код V_sem_split примера 2.20.

Пример 2.21. Решение задачи о читателях и писателях методом передачи эстафеты

<pre> //Количество процессов-читателей, получивших доступ к БД int num_r = 0; //Количество процессов-писателей, получивших доступ к БД int num_w = 0; sem cs = 1, // управляет входом в любое неделимое действие rd = 0, // используется для приостановки читателей wrt = 0; // используется для приостановки писателей int delay_rd = 0; // число приостановленных читателей delay_wrt = 0; // число приостановленных писателей thread Reader(i=0;i<R){ while (true){ //<await(num_w == 0) num_r++;> P(cs); if (num_w > 0){ delay_rd++;V(cs);P(rd); } num_r++; //см. пример 2.20 V_sem_split; Читать базу данных; //<num_r --;> P(cs); num_r--; //см. пример 2.20 V_sem_split; } } </pre>	<pre> thread Writer(j=0;j<W){ while (true) { //<await(num_r == 0 and num_w == 0)num_w++;> P(cs); if (num_r > 0 or num_w > 0) {delay_wrt++;V(cs);P(wrt); } num_w++; //см. пример 2.20 V_sem_split; Записать в базу данных; //<num_w --;> P(cs); num_w--; //см. пример 2.20 V_sem_split; } } </pre>
--	--

Код `V_sem_split` является общим для процедуры сигнализации, его в предыдущей программе можно упрощать, исключая некоторые проверки, поскольку ситуации, в них проверяемые, заведомо не могут случаться в соответствующих частях программы. Например, в конце протокола входа процесса-читателя `num_r>0`, а `num_w=0` всегда, следовательно, первую проверку в коде `V_sem_split` можно упростить, а вторую – и вовсе исключить.

Пример 2.22. Решение задачи о читателях и писателях с помощью семафоров

```
//Количество процессов-читателей, получивших доступ к БД
int num_r = 0;
//Количество процессов-писателей, получивших доступ к БД
int num_w = 0;
sem cs = 1, // управляет входом в любое неделимое действие
    rd = 0, // используется для приостановки читателей
    wrt = 0; // используется для приостановки писателей
int delay_rd = 0; // число приостановленных читателей
    delay_wrt = 0; // число приостановленных писателей
thread Reader(i=0;i<R){
    while (true){
//<await(num_w == 0)
num_r++;>
    P(cs);
    if (num_w > 0) {
        delay_rd++;V(cs);P(rd);
    }
    num_r++;
//Упрощенный код V_sem_split
    if (delay_rd > 0) {
        delay_rd--; V(rd);
    }
    else V(cs);
    Читать базу данных;
//<num_r --;>
    P(cs);
    num_r--;
//Упрощенный код V_sem_split
    if (num_r == 0 and
        delay_wrt > 0){
        delay_wrt--; V(wrt);
    }
    else V(cs);
    }
}

thread Writer(j=0;j<W){
    while (true){
//<await (num_r == 0 and
    num_w == 0) num_w++;>
    P(cs);
    if (num_r>0 or num_w>0){
        delay_wrt++;V(cs);P(wrt);
    }
    num_w++;
//Упрощенный код V_sem_split
    V(cs);
    Записать в базу данных;
// <num_w--;>
    P(cs);
    num_w--;
//Упрощенный код V_sem_split
    if (delay_rd > 0){
        delay_rd--; V(rd);
    }
    else if (delay_wrt > 0){
        delay_wrt--; V(wrt);
    }
    else V(cs);
    }
}
```

Пример 2.22 содержит упрощенный код примера 2.21, в котором исключены все ненужные проверки в протоколах выхода `v_sem_split`. Таким образом, пример 2.22 демонстрирует окончательное решение задачи о читателях и писателях методом передачи эстафеты с помощью семафоров, реализующее преимущество читателей [48].

Из рассмотренных примеров видно, что семафоры – это уникальный низкоуровневый механизм, который можно использовать для синхронизации потоков (как в случае взаимного исключения, так и в случае условного ожидания). Однако конкретная языковая реализация семафоров всегда имеет ряд особенностей. Более того, как правило, существует несколько взаимно дополняющих друг друга механизмов синхронизации. Например, в библиотеке Pthread синхронизацию потоков можно осуществить в той или иной степени с помощью целого набора объектов – мьютексов, семафоров, различных блокировок, барьеров. Интерфейс WinAPI также предоставляет большое разнообразие объектов ядра для синхронизации потоков – критические секции, события, семафоры и т. д. В гл. 3 эти механизмы подробно обсуждаются.

2.6. Мониторы

Семафоры являются фундаментальным механизмом синхронизации. В языковых средствах для многопоточного программирования, как правило, они представлены несколькими реализациями. Вместе с тем семафоры – это низкоуровневый механизм, имеющий ряд недостатков, обычно присущих объектам этого уровня. Например, можно случайно пропустить вызов семафора, защитить не все критические секции или запутаться в разделяемых глобальных по отношению к потокам переменных.

При использовании семафоров две проблемы – взаимное исключение и условная синхронизация – программируются одной и той же парой примитивов, однако это разные понятия, а также хотелось бы иметь разные механизмы их реализации.

Мониторы – программные модули, которые при той же эффективности реализации, что и семафоры, обеспечивают лучшую структуризацию кода. Монитор инкапсулирует представление абстрактного объекта, содержит переменные, хранящие состояние объекта, и процедуры, реализующие операции над ним. Поток получает доступ к переменным в мониторе только путем вызова процедур этого монитора, поэтому взаимное исключение обеспечивается неявно, *два потока не могут одновременно выполняться в одном мониторе, поскольку не могут одновременно выполняться две процедуры монитора*. Идею инкапсуляции данных для управ-

ления доступом к разделяемым переменным в параллельной программе озвучил Э. Дейкстра [16], а мониторы носят имя Хоара, который дал им такое название и предложил с их помощью решение нескольких классических задач.

Параллельная программа, использующая мониторы для взаимодействия и синхронизации, содержит два типа модулей: активные потоки и пассивные мониторы.

Монитор используется, чтобы сгруппировать представление и реализацию разделяемого ресурса (класс ресурса). Монитор состоит из следующих компонент:

- *интерфейс* – предоставляемые ресурсом методы;
- *двоичный семафор* (мьютекс) – используется для обеспечения взаимного исключения. Каждая процедура монитора захватывает мьютекс перед началом работы и удерживает его до своего окончания или вызова функции ожидания на условной переменной (специальной переменной, используемой для синхронизации в мониторе);
- *тело* – переменные, отражающие состояние ресурса.

Для предотвращения состояния гонок логика монитора предполагает наличие некоторого *инварианта*, который должен быть истинен перед любым освобождением мьютекса. Проверка истинности инварианта, как и его наличие, как правило, остается на совести проектировщика монитора.

В общем случае монитор описывается следующим образом:

```
monitor m_name{  
  объявление постоянных переменных  
  операторы инициализации  
  процедуры  
}
```

Процедуры реализуют видимые операции, постоянные переменные разделяются всеми процедурами тела монитора, они существуют, пока существует монитор. Постоянные переменные доступны потокам только через процедуры монитора. Вызов процедуры имеет вид:

```
call monitor_name.operation_name (arguments);
```

Операторы внутри монитора не могут обращаться к переменным, объявленным вне монитора. Постоянные переменные инициализируются до вызова его процедур.

Процедуры монитора *по определению* выполняются со взаимным исключением. Это обеспечивается реализацией языка, библиотекой или ОС, но не программистом. На практике взаимное исключение в языках и библиотеках реализуется с помощью блокировок и семафоров.

Внешний поток вызывает процедуру монитора. Пока выполняются операторы процедуры, она активна. *В любой момент времени может быть активным только один экземпляр только одной процедуры монитора.*

Условная синхронизация в мониторе реализуется специальным механизмом, основой которого является условная переменная – переменная нового типа данных, которые используются *только внутри* монитора.

Условные переменные, как и семафоры, осуществляют блокировку потоков и их пробуждение. Но для семафоров четко определено условие, по которому поток останавливается и возвращается к работе, тогда как условные переменные не содержат никаких условий для управления потоками. Используя условную переменную, можно приостановить выполнение потока, можно «разбудить» один из потоков или все потоки, ожидающие сигнала от данной переменной. Все эти действия осуществляются посредством вызова функций по обработке условной переменной; при этом она не обладает никаким внутренним условием, с которым необходимо считаться при выполнении соответствующих функций.

Условная переменная используется для приостановки работы потока, безопасное выполнение которого пока невозможно. Ожидание длится до перехода монитора в состояние, удовлетворяющее некоторому логическому условию. Значением условной переменной является *очередь приостановленных потоков*, вначале очередь пуста.

Условная переменная объявляется следующим образом:

```
cond cv;
```

Можно объявить массив условных переменных.

```
cond cv_array[n];
```

Программист не может напрямую обращаться к условной переменной. Стандартный набор операций, которые допустимы над условной переменной, приведен в табл. 2.1.

Таблица 2.1

Операция	Краткое описание
<code>empty(cv)</code>	Возвращает «TRUE», если очередь переменной пуста, иначе – «FALSE»
<code>wait(cv)</code>	Ждать в конце очереди
<code>wait(cv, rank)</code>	Ждать в порядке возрастания значения переменной <code>rank</code>
<code>signal(cv)</code>	Запустить поток из начала очереди и продолжить выполнение
<code>signal_all(cv)</code>	Запустить все потоки очереди и продолжить выполнение
<code>minrank(cv)</code>	Возвращает значение ранга потока в начале очереди ожидания

Выполнение потоком операции `wait(cv)` заставляет его перейти в режим ожидания, причем поток ставится в конец очереди ожидания на

переменной `cv`. Чтобы другой приостановленный на этой же переменной поток мог войти в монитор, выполнение операции `wait(cv)` отбирает у потока, вызвавшего ее, исключительный доступ к монитору. Запуск заблокированного на условной переменной `cv` потока происходит при выполнении другим потоком операции `signal(cv)`. При этом если очередь условной переменной пуста, то никакие действия не происходят, иначе запускается поток из начала очереди `cv`.

Существует один неоднозначно разрешимый вопрос: выполняя операцию `signal()`, поток, работая в мониторе, запускает *другой* поток. Возникает дилемма, приводящая к двум возможным вариантам развития событий синхронизации потоков:

- протокол «*сигнализировать и продолжать*» (*S&C*): в мониторе остается выполняющийся поток, а поток, получивший сигнал, переходит в очередь готовых потоков и получает право выполняться при первом удобном случае;

- протокол «*сигнализировать и ожидать*» (*S&W*): управление получает поток из очереди, а сигнализатор переходит в очередь (возможно, в ее начало – протокол «*сигнализировать и срочно ожидать*»).

Таким образом, поток, вызывающий процедуру монитора помещается во входную очередь (если очередь пуста, то поток выполняется сразу). Монитор может освободиться двумя способами: 1) поток, блокирующий монитор, закончил свою работу в мониторе; 2) выполнена операция `wait(cv)`. В любом случае начинает работать поток, первый из входной очереди. Если же работающий поток выполняет процедуру `signal(cv)`, то при стратегии *S&C* поток из начала очереди условной переменной `cv` перемещается в начало очереди монитора, а в мониторе остается сигнализировавший поток. Если же принят протокол *S&W*, то поток, работавший в мониторе, переходит во входную очередь (возможно, в начало), а запускается поток, первый в очереди на условной переменной. Схема синхронизации в мониторах изображена на рис. 2.7.

Первой была предложена процедура «сигнализировать и ожидать» (семантика Хоара). Однако реализация стратегии *S&W* несовместима с тем, что любой процесс (в том числе и выполняющий процедуру монитора) может быть прерван в произвольный момент, и, следовательно, гарантировать истинность условия, при котором была вызвана процедура `signal()`, сложно. В ОС Unix и в библиотеках языков программирования Си и Java принят именно порядок *S&C* (семантика Меса (Mesa)). У такого подхода также есть проблемы, которые надо учитывать при разработке монитора. В частности, сигнализирующий процесс не обязан обеспечивать истинность инварианта перед оповещением, а ожидающий процесс должен повторно проверить условие, которого он дожидается.

В библиотеке PTHREAD монитор не реализован в явном виде, однако механизм условных переменных введен с помощью структуры `pthread_cond_t`. Основное отличие использования условной переменной вне монитора состоит в том, что о взаимном исключении следует заботиться дополнительно. Это происходит, например, путем совместного использования условной переменной и мьютекса для обеспечения взаимного исключения. Монитор поддерживается в языках Ада, С# (и других, поддерживаемых .NET Framework), Concurrent Pascal, Delphi, Java, Squeak Smalltalk и некоторых других.

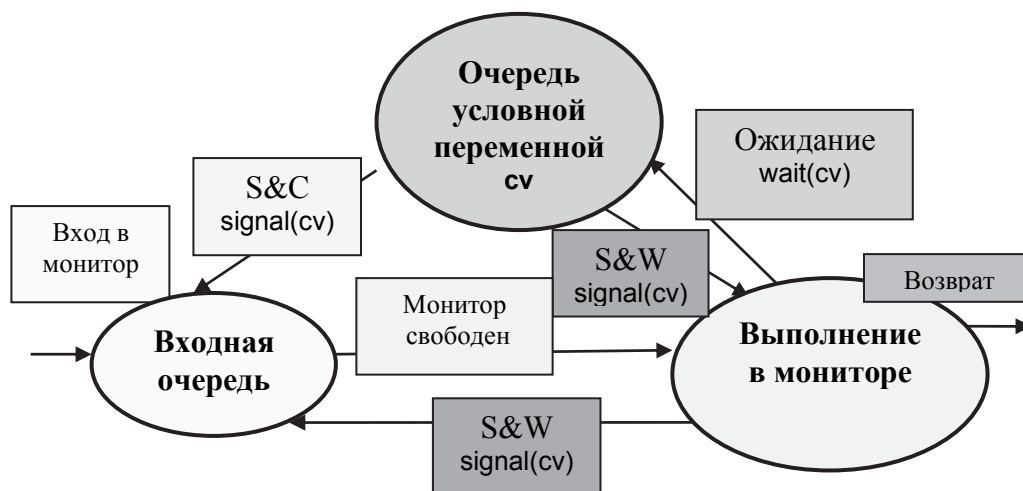


Рис. 2.7. Синхронизация в мониторе

Отсутствие условия на выполнение операций типа `signal()`, на первый взгляд, делает управление потоками более простым. Но эта особенность условных переменных обуславливает их ненадежность, так как поток может быть «разбужен», только если он приостановлен и ожидает сигнала от условной переменной. Таким образом, если сигнал о «пробуждении» был сформирован до «усыпления» потока, то этот сигнал пропадет, поток войдет в состояние «сна» и далее из него не выйдет.

Выше обсуждалась возможность синхронизации процессов по условию с помощью семафоров. Монитор по умолчанию предполагает выполнение своих процедур с взаимным исключением, для программирования условного ожидания используются условные переменные. Продемонстрируем применение условных переменных на задаче о кольцевом буфере, решение которой на основе семафоров рассмотрено ранее.

Пусть по-прежнему буфер представлен массивом `buf[n]`, $n > 1$; `front` – индекс первого сообщения в очереди; `rear` – индекс первой пустой ячейки после сообщения в конце очереди; `count` – количество сообщений в буфере.

Рассмотрим монитор с двумя операциями: `deposit()` – «положить в буфер» и `fetch()` – «извлечь из буфера». Поскольку эти операции по умолчанию происходят с взаимным исключением, то в нашем решении нет никаких аналогов семафоров `mutexP` и `mutexC`. Для реализации условного ожидания вместо семафоров `empty` и `full` используются две условные переменные: `not_empty` получает сигнал, когда `count > 0`, т. е. из буфера можно извлекать данные; `not_full` получает сигнал, когда `count < n`, т. е. в буфер можно положить данные. Задержка на условных переменных происходит в цикле `while`.

Код решения задачи о кольцевом буфере с помощью мониторов содержится в примере 2.23.

Пример 2.23. Код решения задачи о кольцевом буфере с помощью монитора

```
monitor Bounded_Buffer{
    typeT buf[n]; //массив некоторого типа typeT
    int front = 0, //индекс первой заполненной ячейки
        rear = 0, //индекс первой пустой ячейки
        count = 0; //число заполненных ячеек
    cond not_full, //получает сигнал, когда count < n
        not_empty; //получает сигнал, когда count > 0
    procedure deposit(typeT data) {
        while (count == n) wait(not_full);
        buf[rear] = data; rear = (rear+1) % n; count ++;
        signal(not_empty);
    }
    procedure fetch(typeT &result){
        while (count == 0) wait(not_empty);
        result = buf[front]; front = (front + 1) % n; count--;
        signal(not_full);
    }
}
```

В этой главе мы рассмотрели основы многопоточного программирования, получили представление о самых низкоуровневых механизмах реализации синхронизации процессов. Следует отметить, что многие современные языки и среды программирования экранируют от разработчика подробности организации многопоточности, предоставляя свой высокоуровневый API для распараллеливания. Однако каким бы высокоуровневым ни был подход к многопоточному программированию, для написания не только корректных, но и эффективных программ необходимо понимать базовые механизмы многопоточности и синхронизации.

Контрольные вопросы и задания

1. Что такое поток?
2. Чем потоки отличаются от процессов?
3. В чем специфика параллельных программ, взаимодействующих через общую память?
4. Опишите состояния, в которых может находиться поток в системе.
5. Для чего необходима синхронизация потоков? Приведите примеры.
6. Опишите основные схемы взаимодействия потоков: взаимное исключение и условная синхронизация. Ответ проиллюстрируйте ситуационными задачами, для которых постройте схемы взаимодействия потоков.
7. Каким образом используется оператор ожидания для синхронизации потоков?
8. Определите понятие критической секции, опишите общее решение задачи критической секции, перечислите свойства, которым должны удовлетворять протоколы входа и выхода КС.
9. Определите понятия крупно- и мелкомодульного решений. Приведите примеры.
10. Найдите какие-либо алгоритмы с активным ожиданием для решения задачи критической секции, не описанные в главе. Какие достоинства и недостатки каждого из них вы можете назвать?
11. Чем обеспечивается свойство живучести алгоритма поликлиники для решения задачи критической секции?
12. Определите понятие барьера. В чем принципиальное отличие проблемы барьерной синхронизации от проблемы задачи критической секции?
13. Определите понятие флага, опишите принципы использования флагов в барьерной синхронизации.
14. В чем преимущества и недостатки барьеров с управляющим процессом, несимметричных барьеров? Разработайте код для несимметричного барьера с активным ожиданием, основанный на парадигме «взаимодействующие равные».
15. На чем основаны симметричные барьеры? Приведите несколько схем симметричных барьеров. Разработайте крупно- и мелкомодульное решение с активным ожиданием для симметричных барьеров.
16. Определите понятие семафора. Каковы причины, обуславливающие использование семафоров?
17. Как организован семафор?
18. Приведите решение задачи критической секции с использованием семафоров. Обеспечивает ли ваше решение безопасность и живучесть протокола входа? Поясните свой ответ.
19. Напишите обеспечивающее живучесть решение задачи критической секции с помощью семафоров.

20. Поясните использование семафоров для обеспечения безопасного доступа к пересекающемуся множеству ресурсов на примере решения задачи об обедающих философах. Поясните, в каком случае может возникнуть мертвая блокировка. Как этого избежать?

21. Разработайте основанный на семафорах и обладающий свойством живучести алгоритм симметричного решения задачи об обедающих философах.

22. В чем специфика условной синхронизации при использовании семафоров? Назовите принципиальное отличие применения семафоров для условной синхронизации и взаимного исключения.

23. Сделайте сравнительный анализ реализации взаимного исключения и условной синхронизации процессов на примере задачи о кольцевом буфере для n процессов.

24. Дайте понятие разделенного двоичного семафора. Как в общем случае с помощью него реализовать механизм оператора ожидания? Опишите обобщение операций $P()$ и $V()$ для разделенного двоичного семафора.

25. Постройте разделенный двоичный семафор (опишите его структуру и код операций $P()$ и $V()$) для решения задачи о читателях и писателях с преимуществом писателей. Что необходимо изменить в коде примера 2.22 для того, чтобы реализовать преимущество писателей? Измените код примера 2.22 таким образом, чтобы преимущество потоков-читателей реализовалось только, если ожидающих читателей накопилось больше некоторого порогового значения `Crtt_rd`?

26. Постройте разделенный двоичный семафор для решения задачи о читателях и писателях с прохождением транзакций в режиме «грязное чтение»: процесс-читатель может разделять операцию чтения всегда, т.е. даже если доступ к базе данных получил процесс-писатель; процесс-писатель получает доступ к базе данных только в том случае, когда к ней не обращается ни один другой процесс-писатель.

27. Определите понятие монитора. В чем проявляется специфика мониторов? Опишите принцип его функционирования.

28. В каких случаях мониторы оказываются удобнее семафоров?

29. Введите понятие условной переменной. Укажите роль условной переменной в синхронизации процессов.

30. Опишите известные протоколы обработки операции `signal()`.

31. Поясните схему синхронизации процессов в мониторах.

32. Продемонстрируйте отличия и сходства использования семафоров и мониторов на примере решения задачи о кольцевом буфере.

33. Напишите монитор для доступа к базе данных. Укажите, как реализовать преимущество читателей или писателей.

Задачи

2.1–2.4. Напишите решение задач 1.3–1.6 с помощью семафоров. Какие семафоры в решении использованы для реализации взаимного исключения, а какие – для условной синхронизации потоков?

2.5. Напишите код симметричного барьера «бабочка» для $P = 2^s$ потоков, используя:

- механизм активного ожидания;
- семафоры.

2.6. Напишите код симметричного барьера с распространением для любого количества процессов, используя:

- механизм активного ожидания;
- семафоры.

2.7. *Задача о банкете.* На большом банкете разносом блюд занимаются n официантов. Блюда подаются с кухни, при этом к каждому блюду прикреплен номерок с указанием, за какой столик это блюдо следует отнести. Тот официант, который свободен в данный момент, относит блюдо на место. Требуется создать многопоточное приложение, демонстрирующее заказ блюд и работу официантов. При решении использовать парадигму «портфель задач».

2.8. *Задача о парикмахере.* В тихом городке есть парикмахерская. Салон парикмахерской мал, ходить там может только парикмахер и один посетитель. Парикмахер всю жизнь обслуживает посетителей. Когда в салоне никого нет, он спит в кресле. Когда посетитель приходит и видит спящего парикмахера, он будит его, садится в кресло и спит, пока парикмахер занят стрижкой. Если посетитель приходит, а парикмахер занят, то он встает в очередь и засыпает. После стрижки парикмахер сам провожает посетителя. Если есть ожидающие посетители, то парикмахер будит первого в очереди и ждет, пока тот сядет в кресло парикмахера и начинает стрижку. Если никого нет, парикмахер садится в свое кресло и засыпает до прихода посетителя. Создать многопоточное приложение, моделирующее рабочий день парикмахерской с помощью семафоров.

2.9. *Первая задача о Винни-Пухе, или «Правильные пчелы».* В одном лесу живут n пчел и один медведь Винни, которые пользуются, каждый по своему, одним горшком меда вместимостью N глотков. Сначала горшок пустой. Пока горшок не наполнится, медведь спит. Как только горшок заполняется, Винни просыпается и съедает весь мед, после чего снова засыпает. Каждая пчела многократно собирает по одному глотку меда и кладет его в горшок. Пчела, которая приносит последнюю порцию меда, наполняющую горшок доверху, будит медведя. Создать многопоточное приложение, моделирующее поведение пчел и медведя.

2.10. *Вторая задача о Винни-Пухе, или «Неправильные пчелы».* Семья неправильных пчел, подсчитав в конце месяца убытки от наличия в лесу Винни-Пуха, решила разыскать его и наказать в назидание всем другим любителям сладкого. Для поисков медведя семья поделила лес на участки, каждый из которых прочесывает одна группа неправильных пчел. В случае нахождения медведя на своем участке группа сообщает всем остальным о местонахождении виновника, после чего вся семья проводит показательное наказание бедного Винни и возвращается в улей. Если участок прочесан, а Винни-Пух на нем не обнаружен, группа возвращается в улей и ждет сигнала от других пчел. Ничего не подозревающий о планах пчел Винни гуляет по лесу, иногда укладываясь как следует подремать. Требуется создать многопоточное приложение, моделирующее действия пчел и медведя. При распределении леса между группами пчел использовать парадигму портфеля задач. Необходимо предусмотреть ситуацию, когда из-за случайного блуждания Винни ни одна группа пчел его не нашла. В этом случае все пчелы, вернувшись в улей и отдохнув, вновь отправляются на поиски медведя.

Глава 3 | УПРАВЛЕНИЕ ПОТОКАМИ С ПОМОЩЬЮ ФУНКЦИЙ WinAPI

В главе рассмотрены основы многопоточного программирования с помощью функций WinAPI. Дается общее понятие объекта ядра ОС Windows, процесса и потока. Обсуждаются особенности реализации проблемы взаимного исключения потоков одного процесса, т. е. находящихся в общем адресном пространстве (синхронизация в пользовательском режиме), и общий случай синхронизации потоков, относящихся к разным процессам, с помощью объектов ядра. Рассмотрен один из простых механизмов организации связи между потоками разных процессов, например, для обмена данными.

3.1. Объекты ядра

Основные понятия

Операционная система Windows позволяет оперировать несколькими типами объектов ядра, такими как процессы, потоки, проекции файлов, события, семафоры, мьютексы, каналы и т. д.

Каждый объект ядра – это блок памяти, выделенный ядром и доступный только ядру. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (имя объекта, дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Например, у объекта «процесс» есть идентификатор процесса, базовый приоритет и код завершения, а у объекта «файл» – смещение в байтах, режим разделения и режим открытия.

Для создания объекта ядра необходимо в программе вызвать функцию Win32 API. Например, `CreateFileMapping()` заставляет систему сформировать объект «проекция файла» (file-mapping object).

Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение введено намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет вводить, удалять или изменять элементы структур ОС, не нарушая работы каких-либо приложений.

Каким же образом приложение может оперировать объектами ядра? Win32 API предусматривает набор функций, обрабатывающих объекты ядра

по строго определенным правилам. Доступ к объектам ядра может быть получен только через эти функции.

Функция создания объекта ядра всегда возвращает *описатель*, идентифицирующий созданный объект. Описатель следует рассматривать как «непрозрачное» 32-битное значение (в программах Си это переменная типа `HANDLE`), которое может быть использовано любым потоком процесса. Приложение передает описатель объекта ядра Win32-функциям, сообщая системе, какой объект ядра его интересует.

Для бóльшей надежности ОС значения *описателей* зависят от конкретного процесса. Даже если с помощью какого-либо механизма меж-процессной связи описатель объекта, созданного в процессе «1», передать потоку процесса «2», то любой вызов из процесса «2» со значением полученного описателя даст ошибку.

Объекты ядра принадлежат ядру, а не процессу. Иначе говоря, если процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен. Это происходит, если созданный одним процессом объект ядра используется другим процессом. Ядро запретит разрушение объекта до тех пор, пока от него не откажутся все пользующиеся объектом процессы.

Одним из элементов данных любого объекта ядра является счетчик его пользователей. В момент создания объекта счетчику присваивается единица. При открытии существующего объекта ядра каждым новым процессом счетчик увеличивается на единицу. При завершении процесса счетчики всех объектов ядра, которые использовал этот процесс, автоматически уменьшаются на единицу. Как только счетчик какого-либо объекта обнуляется, ядро ОС уничтожает этот объект.

Таблица описателей

При инициализации процесса система создает в нем таблицу описателей, используемую только для объектов ядра [8, 34]. Сведения о структуре этой таблицы и управлении ею не задокументированы. На рис. 3.1 представлена примерная структура таблицы описателей.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа	Флаги
--------	--	------------------	-------

Рис. 3.1. Примерная структура таблицы описателей объектов ядра

При инициализации процесса таблица описателей пуста. При вызове одним из потоков процесса Win32 API функции, создающей объект ядра (например, `CreateFileMapping()`), ядро выделяет для этого объекта блок памяти и инициализирует его; далее ядро просматривает таблицу описате-

лей, принадлежащую данному процессу, и отыскивает свободную запись. Поскольку таблица еще пуста, ядро обнаруживает структуру с индексом 1 и инициализирует ее. Указатель устанавливается на внутренний адрес памяти структуры данных объекта ядра, маска доступа – на доступ без ограничений, и, наконец, определяется последний компонент – флаги.

Все функции, создающие объекты ядра, возвращают описатели (HANDLE), которые привязаны к конкретному процессу и могут быть использованы в его любом потоке. Значение описателя представляет собой индекс в таблице описателей, принадлежащей процессу, следовательно, идентифицирует место, где хранится информация, связанная с объектом ядра.

Независимо от того, как именно создан объект ядра, по окончании работы с ним его нужно закрыть вызовом Win32 API функции `CloseHandle()`:

```
BOOL CloseHandle(HANDLE hobj);
```

Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей описатель объект, к которому этот процесс действительно имеет доступ. Если переданный описатель неверен, функция возвращает `FALSE`, а функция `GetLastError()` – код `ERROR_INVALID_HANDLE`. Если же индекс достоверен, система получает адрес структуры данных объекта ядра и уменьшает в этой структуре счетчик числа пользователей. Как только счетчик обнулится, ядро удалит объект из памяти. Перед самым возвратом управления `CloseHandle()` удаляет соответствующую запись из таблицы описателей, после этого данный описатель недействителен в вызвавшем закрытие процессе, использовать его нельзя. Запись из таблицы описателей процесса удаляется независимо от того, разрушен объект ядра или нет. После вызова `CloseHandle()` процесс больше не получит доступ к этому объекту ядра.

Совместное использование объектов ядра несколькими процессами

Существует необходимость в совместном использовании объектов ядра потоками, исполняемыми в разных процессах. Например:

- объекты «проекции файлов» (file-mapping object) позволяют двум процессам, исполняемым на одном компьютере, совместно использовать одни и те же блоки данных;
- «почтовые ящики» (mailslots) и именованные каналы (named pipes) дают возможность обмениваться данными процессам, исполняемым на разных компьютерах в Сети;

- мьютексы (mutexes), семафоры (semaphores) и события (events) позволяют синхронизировать потоки, исполняемые в разных процессах, чтобы одно приложение могло уведомить другое об окончании той или иной операции.

Но описатели объектов ядра имеют смысл только в конкретном процессе, поэтому разделение объектов ядра между несколькими процессами в Win32 – задача весьма непростая.

У разработчиков ОС было несколько веских причин сделать описатели «процессозависимыми», основными считаются устойчивость операционной системы к сбоям и защита. Объекты ядра защищены, процесс, прежде чем оперировать с объектом, должен запросить разрешение на доступ к нему у ОС. Процесс-создатель объекта может предотвратить несанкционированный доступ к этому объекту со стороны другого процесса.

Существует три способа разделения объектов ядра.

1. *Наследование.* Наследование применимо, только если процессы связаны «родственными» отношениями (родительский – дочерний). Например, родительскому процессу доступны один или несколько описателей объектов ядра, и он решает, породив дочерний процесс, передать ему по наследству доступ к своим объектам ядра.

При вызове функции создания объекта в качестве одного из ее параметров передается структура `LPSECURITY_ATTRIBUTES`, которая определяет среди прочего и наследуемость объекта. Если при создании дочернего процесса в параметрах функции `CreateProcess()` флаг наследования объектов `blnheritHandles` выставлен в значение `TRUE`, то этот дочерний процесс может наследовать объекты. ОС создает дочерний процесс, но не дает ему немедленно начать свою работу. Сформировав в нем, как обычно, новую (пустую) таблицу описателей, ОС считывает таблицу родительского процесса и копирует ее записи в таблицу дочернего, причем в те же позиции. Последний факт чрезвычайно важен, так как означает, что описатели будут идентичны в обоих процессах (родительском и дочернем).

Кроме того, ОС увеличивает значения счетчиков соответствующих объектов ядра, поскольку эти объекты теперь используются обоими процессами.

Сразу после возврата управления функцией `CreateProcess()` родительский процесс может закрыть свой описатель объекта, и это никак не отразится на способности дочернего процесса манипулировать этим объектом. Чтобы уничтожить какой-то объект ядра, его описатель должны закрыть (вызовом `CloseHandle()`) оба процесса – родительский и дочерний.

Следует отметить, что наследуются только описатели объектов, существующие на момент создания дочернего процесса. Если родительский

процесс создаст после этого новые объекты ядра с наследуемыми описателями, то эти описатели дочернему процессу уже недоступны.

2. *Именованные объекты.* Второй способ разделения объектов ядра процессами связан с возможностью присвоения имени объекту ядра. Именовывать можно многие, но не все виды объектов. Именуются, к примеру, мьютексы, события, семафоры и проекции файлов.

В качестве имени объекта передается строка, завершающаяся нулевым символом, длиной не более 260 символов и не содержащая символов «/». Все именованные объекты разделяют общее для системы пространство имен, поэтому нельзя создать два разнотипных объекта с одинаковым именем.

Пусть два разных процесса вызовут функцию вида `CreateObject()`, указав для объектов *одного типа* одинаковые имена. Для того, кто это сделал первым (например, процесс А), будет отработана обычная процедура создания объекта. Для второго процесса (соответственно, В) система проверит и убедится, что объект ядра с таким же именем и такого же типа существует. При этом система считает вызов `CreateObject()` из процесса В успешным и создает в его таблице описателей новую запись, а затем инициализирует ее так, чтобы она указывала на уже существующий объект ядра.

Надо помнить, что вызов процессом В функции `CreateObject()` не привел к созданию нового объекта. Процесс В получил свой описатель, который идентифицирует уже существующий объект. Счетчик объекта, конечно же, увеличился на единицу, и теперь он не разрушится, пока его описатели не закроют оба процесса А и В. Скорее всего, значения описателей этого объекта в обоих процессах разные, но так и должно быть: каждый процесс будет оперировать с данным объектом ядра, используя свой описатель.

Параметры защиты созданного таким образом объекта будут соответствовать параметрам, переданным процессом А.

Отметим, что если тип объекта, создаваемого процессом В, *не совпадает* с уже созданным объектом с тем же именем, то система сгенерирует ошибку, и второй объект не будет создан.

Вместо `Create`-функций возможно использование одной из `Open`-функций. Все `Open`-функции имеют общий прототип:

```
BOOL OpenHandle(DWORD dwDesiredAccess,
                  BOOL blnheritHandle,
                  LPCTSTR IpszName);
```

Функция `OpenHandle()` просматривает единое пространство имен объектов ядра, пытаясь найти совпадение. Если объекта ядра с указанным в параметрах функции именем нет или он другого типа, `Open`-функции воз-

вращают `NULL`. Но если объект ядра с заданным именем существует и если его тип идентичен тому, что указан, система проверяет, разрешен ли к данному объекту доступ запрошенного вида. Если доступ разрешен, таблица описателей в вызывающем процессе обновляется, и счетчик числа пользователей объекта возрастает на единицу. Опишем параметры функции.

Параметр `DWORD dwDesiredAccess` определяет вид запрашиваемого к объекту доступа. Для каждого объекта ядра имеется свой набор флагов, любая комбинация которых и передается в этом параметре. Доступ может запрашиваться полный или на выполнение над объектом определенных операций.

Параметр `BOOL blnheritHandle` обозначает свойство наследования. Если значение этого параметра `TRUE`, то будет получен наследуемый описатель.

Параметр `LPCTSTR lpszName` определяет имя объекта ядра. Он всегда должен содержать адрес строки с нулевым символом в конце (передать `NULL` нельзя).

Главное отличие между вызовом `Create`- и `Open`-функций в том, что при отсутствии указанного объекта `Create`-функция создает его, а `Open`-функция просто сообщает об ошибке.

3. *Дублирование описателей объектов.* Последний механизм совместного использования объектов ядра несколькими процессами – создание дубликата описателя объекта:

```
BOOL DuplicateHandle (HANDLE hSourceProcessHandle,  
                      HANDLE hSourceHandle,  
                      HANDLE hTargetProcessHandle,  
                      LPHANDLE lpTargetHandle,  
                      DWORD dwDesiredAccess,  
                      BOOL bInheritHandle, DWORD dwOptions);
```

Эта функция создает копию записи из таблицы описателей процесса `hSourceProcessHandle` в таблице процесса `hTargetProcessHandle`, не обязательно вызвавшего эту функцию. Опишем параметры функции.

Параметр `HANDLE hSourceProcessHandle` задает специфичный для вызывающего процесса описатель объекта ядра типа «процесс», который является владельцем запрашиваемого объекта.

Параметр `HANDLE hSourceHandle` задает описатель запрашиваемого объекта ядра любого типа, специфичного для процесса, на который указывает параметр `hSourceProcessHandle`.

Параметр `HANDLE hTargetProcessHandle` задает специфичный для вызывающего процесса описатель объекта ядра типа «процесс», в таблицу которого копируется запись (процесс-приемник).

Параметр `LPHANDLE IpTargetHandle` задает адрес переменной типа `HANDLE`, в который *возвращается* индекс записи с копией описателя из процесса-источника, специфичного для процесса, идентифицируемого параметром `hTargetProcessHandle`.

Параметр `DWORD dwDesiredAccess` задает маску доступа, устанавливаемую для данного описателя объекта ядра в процессе-приемнике.

Параметр `BOOL PInheritHandle` задает флаг наследования, устанавливаемый для данного описателя объекта ядра в процессе-приемнике.

Параметр `DWORD dwOptions` задает дополнительные параметры дублирования описателя. Параметр может быть равен 0 или любой комбинации двух флагов – `DUPLICATE_SAME_ACCESS` и `DUPLICATE_CLOSE_SOURCE`.

Флаг `DUPLICATE_SAME_ACCESS` сигнализирует о том, что у описателя, получаемого процессом-приемником, должна быть та же маска доступа, что и у описателя в процессе-источнике. Таким образом, этот флаг заставляет `DuplicateHandle()` игнорировать параметр `dwDesiredAccess`. Флаг `DUPLICATE_CLOSE_SOURCE` приводит к закрытию описателя в процессе-источнике. Он позволяет процессам обмениваться объектом ядра как эстафетной палочкой. При этом счетчик объекта не меняется:

```
BOOL CloseHandle(HANDLE hObj) .
```

Функция закрывает переданный ей описатель объекта ядра. Желательно выполнять функцию для всех объектов, созданных в потоке перед его завершением.

3.2. Процессы

Процесс обычно определяют как экземпляр выполняемой программы.

В Win32 процессу отводится 4 Гб адресного пространства. Win32-процесс инертен, т. е. ничего не исполняет, а просто владеет 4 Гб памяти, содержащей код и данные `exe`-файла приложения. В это же пространство загружаются код и данные `DLL`-библиотек, если того требует `exe`-файл. Кроме адресного пространства процессу принадлежат такие ресурсы, как файлы, динамически выделяемые области памяти и потоки. Ресурсы, создаваемые при жизни процесса, обязательно уничтожаются при его завершении.

Чтобы процесс что-нибудь выполнил, в нем нужно создать поток. Каждый процесс должен содержать как минимум один поток. Именно потоки отвечают за исполнение кода, содержащегося в адресном пространстве процесса. Один процесс может владеть несколькими потоками, которые

«одновременно» исполняют код в адресном пространстве процесса. Для этого каждый поток должен располагать собственным набором регистров процессора и собственным стеком.

Чтобы все потоки работали, ОС отводит каждому из них определенное процессорное время. Выделяя потокам отрезки времени (называемые квантами) по принципу карусели, ОС создает тем самым иллюзию одновременного выполнения множества потоков.

В создании процессов и управлении ими есть большое количество тонкостей, связанных с особенностями ОС Windows. Их подробное рассмотрение выходит за рамки обсуждаемой темы. Далее обсуждаются только основные моменты существования процесса и потока в ОС MS Windows.

Процесс создается при вызове приложением следующей функции:

```
BOOL CreateProcess (
    LPCTSTR IpszApplicationName,
    LPCTSTR IpszCommandLine,
    LPSECURITY_ATTRIBUTES IpsaProcess,
    LPSECURITY_ATTRIBUTES IpsaThread,
    BOOL flnherithandles,
    DWORD fdwCreate,
    LPVOID IpvEnvironment,
    LPTSTR IpszCurDir,
    LPSTARTUPINFO IpsiStartInfo,
    LPPROCESS_INFORMATION IppiProcInfo);
```

При вызове потоком функции `CreateProcess()` ОС создает объект ядра «процесс» с начальным значением счетчика числа его пользователей, равным 1. Этот объект не сам процесс, а компактная структура данных, через которую ОС управляет процессом. Таким образом, объект ядра «процесс» следует рассматривать как структуру данных со статистической информацией о процессе.

Затем система создает для нового процесса виртуальное адресное пространство размером 4 Гб и загружает в него код и данные как для исполняемого файла, так и всех DLL (если таковые требуются).

Далее система формирует объект ядра «поток» для первичного потока нового процесса. Как и в первом случае, объект ядра «поток» – это компактная структура данных, через которую ОС управляет потоком.

Первичный поток начнет с исполнения стартового кода из стандартной библиотеки Си, который вызовет функцию `WinMain()` в программе (или `main()`, если приложение относится к консольному типу).

Если системе удастся создать новый процесс и его первичный поток, `CreateProcess()` вернет `TRUE`.

Рассмотрим параметры функции `CreateProcess()`.

Параметр `IpszApplicationName` определяет имя исполняемого файла, которым будет пользоваться новый процесс.

Параметр `IpszCommandLine` определяет полную командную строку, передаваемую системой создаваемому процессу.

Если параметр `IpszApplicationName` равен `NULL`, то `CreateProcess()` анализирует строку по адресу `IpszCommandLine`, извлекая первый компонент и полагая, что это имя исполняемого файла, который необходимо запустить. Если в имени файла не указано расширение, она считает его `exe`. Далее функция приступает к поиску заданного файла и делает это в следующем порядке: каталог, содержащий `exe`-файл вызывающего процесса; текущий каталог вызывающего процесса; системный каталог `Windows`; основной каталог `Windows`; каталоги, перечисленные в переменной окружения `PATH`.

Конечно, если в имени файла указан полный путь доступа, ОС сразу обращается туда и не просматривает упомянутые каталоги.

Найдя нужный исполняемый файл, ОС создает новый процесс и проецирует код и данные исполняемого файла на адресное пространство этого процесса.

Затем система обращается к процедурам стартового кода из стандартной библиотеки Си. Процедура стартового кода, в свою очередь, анализирует командную строку процесса и передает `WinMain()` адрес первого (за именем исполняемого файла) аргумента как `IpszCmdLine`.

Если параметр `IpszApplicationName` содержит адрес строки, то вместе с именем исполняемого файла, который надо запустить, необходимо указать и расширение, и полный путь, поскольку в этом случае имя не дополняется расширением `exe` автоматически. Если полный путь не задан, `CreateProcess()` предполагает, что файл находится в текущем каталоге. Если в текущем каталоге файла нет, функция не станет искать его в других каталогах и `CreateProcess()` завершится с ошибкой.

Но даже при указанном в `IpszApplicationName` имени файла `CreateProcess()` все равно передает новому процессу содержимое параметра `IpszCommandLine` как командную строку.

Следующие три параметра функции `CreateProcess()` задают атрибуты защиты, с которыми порождается новый поток.

Параметр `IpsaProcess` определяет атрибуты защиты для объекта «процесс», *параметр* `IpsaThread` — атрибуты защиты для объекта «поток», который создается для процесса первым (первичный поток). *Параметр* `flInheritHandles` определяет, будут ли переданы дочернему процессу все наследуемые описатели (`TRUE`).

Если параметры `IpsaProcess` и `IpsaThread` содержат `NULL`, то система закрепит за соответствующими объектами (процесс и поток) дескрипторы защиты, принятые по умолчанию. В качестве альтернативы можно объявить и инициализировать две структуры `SECURITY_ATTRIBUTES`, тем самым создавая объекты «процесс» и «поток» с собственными атрибутами защиты.

Структуры `SECURITY_ATTRIBUTES` для параметров `IpsaProcess` и `IpsaThread` используются и при объявлении соответствующих объектов, наследуемых любым дочерним процессом. При этом параметр `fInheritHandles` должен быть равен `TRUE`, тогда дочернему процессу будут переданы все наследуемые описатели.

Параметр `fdwCreate` определяет флаги, влияющие в подробностях на то, как именно создается новый процесс. Несколько флагов комбинируются с помощью булева оператора `OR`. Возможные значения флагов можно посмотреть, например, в [34].

При создании процесса можно задать и класс его приоритета. Однако это необязательно и даже, как правило, не рекомендуется; система присваивает новому процессу класс приоритета по умолчанию. Существующие классы приоритетов перечислены в табл. 3.1.

Таблица 3.1

Класс приоритета	Обозначение
Idle (простаивающий)	IDLE_PRIORITY_CLASS
Normal (нормальный)	NORMAL_PRIORITY_CLASS
High (высокий)	HIGH_PRIORITY_CLASS
Realtime (реального времени)	REALTIME_PRIORITY_CLASS

Параметр `IpvEnvironment` указывает на блок памяти, хранящий строки переменных окружения, которыми будет пользоваться новый процесс. Обычно вместо `IpvEnvironment` передается `NULL`, в результате чего дочерний процесс наследует строки переменных окружения от родительского процесса. В качестве альтернативы можно вызвать функцию `GetEnvironmentStrings()`.

```
LPVOID GetEnvironmentStrings(VOID)
```

Функция позволяет узнать адрес блока памяти со строками переменных окружения, используемых вызывающим процессом. Полученный адрес можно занести в параметр `IpvEnvironment` функции `CreateProcess` (именно это и делает `CreateProcess()`, если передать ей `NULL` вместо `IpvEnvironment`).

Параметр `IpszCurDir` позволяет родительскому процессу установить текущие диск и каталог для дочернего процесса. Если его значение `NULL`, рабочий каталог нового процесса будет расположен там же, где и у приложения, его породившего. А если он отличен от `NULL`, то должен указывать на строку (с нулевым символом в конце), содержащую нужный диск и каталог. В путь надо включать и букву диска.

Параметр `lpSiStartInfo` указывает на структуру `STARTUPINFO`, элементы которой используются Win32-функциями при создании нового процесса.

Большинство приложений порождают процессы с атрибутами по умолчанию. Но и в этом случае надо как минимум инициализировать все элементы структуры `STARTUPINFO` нулевыми значениями, а в элемент `si` занести размер этой структуры, что делается с помощью следующей строки:

```
STARTUPINFO si = sizeof (si);
```

Тогда при создании процесса передается адрес структуры `STARTUPINFO`:

```
CreateProcess(..., &si, ...);
```

Параметр `IppiProcInfo` указывает на структуру `PROCESS_INFORMATION`, которую следует предварительно создать. Структура имеет следующий вид:

```
typedef struct _PROCESS_INFORMATION{
    HANDLE hProcess;
    HANDLE hThread;
    DWORD  dwProcessId;
    DWORD  dwThreadId;
} PROCESS_INFORMATION;
```

Ее элементы инициализируются самой функцией `CreateProcess()`.

Создание нового процесса влечет за собой и создание объектов ядра «процесс» и «поток». В момент создания система присваивает счетчику каждого объекта начальное значение «1».

Далее, перед самым возвратом управления, `CreateProcess()` открывает объекты «процесс» и «поток», заносит их описатели, специфичные для данного процесса, в элементы `hProcess` и `hThread` структуры `PROCESS_INFORMATION`. Когда `CreateProcess()` открывает эти объекты, счетчики каждого из них увеличиваются до двух.

Это означает, что перед тем как система сможет высвободить из памяти объект «процесс», процесс должен быть завершен (счетчик уменьшается до 1), а родительский процесс – вызвать функцию `CloseHandle()` (и тем самым обнулить счетчик). То же самое относится и к объекту

«поток»: поток должен быть завершен, а родительский процесс должен закрыть описатель объекта «поток».

Созданному процессу присваивается уникальный идентификатор: никакие два процесса, выполняемые в системе, не могут иметь одинаковые идентификаторы. То же касается и потоков.

Завершая свою работу, `CreateProcess()` заносит значения идентификаторов в элементы `dwProcessId` и `dwThreadId` структуры `PROCESS_INFORMATION`. Используя их, родительский процесс может обращаться к дочернему.

Система способна повторно использовать идентификаторы процессов и потоков. Например, при создании процесса система формирует объект «процесс», присваивая ему идентификатор со значением `0x22222222`. Создавая новый объект «процесс», система уже не присвоит ему данный идентификатор. Но после выгрузки из памяти первого объекта следующему создаваемому объекту «процесс» может быть присвоен тот же идентификатор `0x22222222`. Это может привести к ошибкам в работе приложений, поскольку в некоторых случаях система может закрыть объект ядра, несмотря на то, что им кто-то пользуется. В этом случае описатель объекта ядра будет указывать на несуществующий процесс или другой процесс, имеющий тот же идентификатор.

Зная, что значение счетчика выше нуля, можно свободно оперировать идентификатором процесса. А когда необходимость в нем отпадет, следует вызвать `CloseHandle()`, чтобы уменьшить счетчик объекта «процесс». Затем стоит удостовериться, что этот идентификатор больше нигде не используется.

Процесс можно завершить одним из трех способов:

1. Один из потоков процесса вызывает функцию `ExitProcess()`.
 2. Поток другого процесса вызывает функцию `TerminateProcess()` (нежелательный способ).
 3. Все потоки процесса завершаются самостоятельно.
- Рассмотрим функции завершения подробнее.

```
VOID ExitProcess(UINT fuExitCode);
```

Функция завершает процесс и заносит в параметр `fuExitCode` код завершения процесса. Эта функция должна быть вызвана одним из потоков процесса. Возвращаемого значения у `ExitProcess()` нет, так как результат ее действия – завершение процесса. Если за вызовом этой функции в программе присутствует какой-нибудь код, он никогда не исполняется.

Вызов `ExitProcess()` – самый распространенный способ завершения процесса, поскольку эта функция вызывается автоматически в тот момент, когда поток `WinMain()` в программе возвращает управление старто-

вому коду из стандартной библиотеки Си. Стартовый код обращается к `ExitProcess()`, передавая ей значение, возвращенное `WinMain()`. При завершении процесса прекращается выполнение и всех его потоков.

```
BOOL TerminateProcess(HANDLE hProcess, DINT fuExitCode);
```

Функция завершает процесс. Главное отличие этой функции от `ExitProcess()` в том, что ее может вызвать любой поток и завершить любой процесс. *Параметр* `hProcess` идентифицирует описатель завершаемого процесса, а в *параметр* `fuExitCode` помещается в код завершения процесса.

Пользоваться `TerminateProcess()` не рекомендуется; к ней можно прибегнуть лишь в том случае, когда иным способом процесс завершить не удастся. При нормальном ходе вещей система уведомляет о завершении процесса все связанные с ним DLL-модули. Поскольку этого не происходит, если процесс завершается вызовом `TerminateProcess()`, то не исключено некорректное завершение процесса.

Например, программа может задействовать DLL-модуль, который при отключении от процесса сбрасывает данные из какого-то буфера в дисковый файл. В обычных условиях отключение DLL происходит при его выгрузке с помощью функции `FreeLibrary()`. Поскольку при обращении к `TerminateProcess()` DLL-модуль об отключении не уведомляется, он не сможет выполнить своей задачи.

Заметим, при любом способе завершения процесса система гарантирует освобождение занятой процессом памяти и объектов `User` или `GDI`, а также закрытие всех открытых файлов и уменьшение счетчиков объектов ядра.

Если *все потоки процесса завершаются самостоятельно* (если все потоки вызвали `ExitThread()` или их закрыли вызовом `TerminateThread()`), операционная система больше не считает нужным «содержать» адресное пространство данного процесса. Обнаружив, что в процессе не исполняется ни один поток, она немедленно завершает его. При этом код завершения процесса приравнивается к коду завершения последнего потока.

При завершении процесса система совершает следующие действия:

1. Выполнение всех потоков в процессе прекращается.
2. Все объекты `User` и `GDI`, созданные процессом уничтожаются, а объекты ядра закрываются.
3. Объект ядра «процесс» переходит в свободное, или незанятое (signaled), состояние. Прочие потоки в системе могут приостановить свое выполнение вплоть до завершения данного процесса.

4. Код завершения процесса меняется со значения `STILL_ACTIVE` на код, переданный в `ExitProcess()` или `TerminateProcess()`.

5. Счетчик объекта ядра «процесс» уменьшается на 1.

Связанный с завершаемым процессом объект ядра «процесс» не освобождается, пока не будут закрыты все ссылки на него. Кроме того, закрытие процесса не приводит к автоматическому завершению порожденных им процессов.

По завершении процесса его код и выделенные ему ресурсы удаляются из памяти. Однако закрытая память, выделенная системой для объекта ядра «процесс», не освобождается, пока счетчик числа его пользователей не достигнет нуля. А это произойдет, только если все прочие процессы, создавшие или открывшие описатели для завершаемого процесса, уведомят систему (вызовом `CloseHandle()`) о том, что ссылки на этот процесс им больше не нужны.

3.3. Потоки

Поток определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Начинаясь со стартового кода из стандартной библиотеки Си (C++) (который, в свою очередь, вызывает функцию `WinMain()` для оконных и функцию `main()` для консольных приложений), он существует до того момента, когда `WinMain()` (`main()`) возвращает управление стартовому коду и тот вызывает функцию `ExitProcess()`. Большинство приложений обходится единственным, первичным, потоком. Однако процессы могут создавать дополнительные потоки, что позволяет добиваться минимального простоя процессора и работать эффективнее.

Каждому потоку выделяется собственный стек в адресном пространстве процесса. При использовании *статических* и *глобальных* переменных не исключена возможность одновременного незащищенного обращения к ним из нескольких потоков, что может вызвать проблемы синхронизации, рассмотренные в гл. 1–2. *Локальные* и *автоматические* переменные создаются в стеке потока, следовательно, они свободны от проблем синхронизации.

У каждого потока – собственный набор регистров процессора, называемый контекстом потока. Эта структура с именем `CONTEXT` отражает состояние регистров процессора на момент последнего исполнения потока. Структура `CONTEXT` является единственной структурой данных в Win32, зависимой от типа процессора. В справочном файле по Win32 она, к сожалению, не описана. Однако в файле `WINNT.H` находится несколько определений `CONTEXT` (для x86, MIPS, Alpha и PowerPC). Компилятор сам выбирает нужный вариант структуры в зависимости от типа процессора, для которого предназначен `exe`- или `DLL`-модуль.

При выделении потоку очередного кванта процессорного времени ОС инициализирует регистры процессора содержимым структуры `CONTEXT` и специальный регистр – указатель команд – идентифицирует адрес следующей машинной команды, необходимой для выполнения потока. В структуру `CONTEXT` включен также указатель стека, который определяет адрес стека, принадлежащего потоку.

Как было сказано выше, при создании процесса создается также и его первичный поток. Если же есть необходимость, чтобы первичный поток породил дополнительные потоки, нужно воспользоваться функцией `CreateThread()`.

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES Ipsa,
    DWORD cbStack,
    LPTHREAD_START_ROUTINE IpStartAddr,
    LPVOID IpvThreadParm,
    DWORD fdwCreate,
    LPDWORD IpIDThread);
```

При каждом вызове этой функции система выполняет следующую последовательность действий:

1. Создает объект ядра «поток» для идентификации и управления «новорожденным» потоком. В нем хранится большая часть системной информации, необходимой для управления потоком. Описатель этого объекта – значение, возвращаемое функцией `CreateThread()`.

2. Инициализирует код завершения потока (регистрируемый в объекте ядра «поток») идентификатором `STILL_ACTIVE` и присваивает счетчику простоя потока (thread's suspend count) единицу. Последний тоже запоминается в объекте ядра «поток».

3. Создает для нового потока структуру `CONTEXT`.

4. Формирует стек потока, для чего резервирует область в адресном пространстве, передает ему 2 страницы физической памяти и присваивает им атрибут защиты `PAGE_READWRITE`, а второй странице (если считать снизу вверх) дополнительно устанавливает атрибут `PAGE_GUARD`.

5. Помещает значения `IpStartAddr` и `IpvThreadParm` в самый верх стека.

6. Инициализирует регистры – указатель стека и указатель команд – в структуре `CONTEXT` потока.

Рассмотрим параметры функции `CreateThread()`

Параметр `Ipsa` является указателем на структуру `SECURITY_ATTRIBUTES`. Если требуется, чтобы объекту «поток» были присвоены атрибуты защиты по умолчанию, следует передать в этом параметре `NULL`.

А чтобы дочерние процессы смогли наследовать описатель данного объекта «поток», требуется определить структуру `SECURITY_ATTRIBUTES` и инициализировать ее элемент `bInheritHandle` значением `TRUE`.

Параметр `cbStack` определяет, какую часть адресного пространства поток сможет использовать под свой стек. Заметим, что функция `CreateProcess()`, запуская приложение, вызывает функцию `CreateThread()` для первичного потока. При этом `CreateProcess()` заносит в параметр `cbStack` значение, хранящееся в самом исполняемом файле. Управлять этим значением позволяет ключ `/STACK` компоновщика:

```
/STACK:[reserve] [, commit]
```

Аргумент `reserve` определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию 1 Мб). Аргумент `commit` задает объем физической памяти, изначально передаваемой зарезервированному региону стека (по умолчанию 1 страницу). По мере исполнения кода в потоке, весьма вероятно, понадобится отвести под стек больше одной страницы памяти. При переполнении стека возникнет исключение. Перехватив это исключение, система передаст зарезервированному пространству еще одну страницу (или столько, сколько указано в аргументе `commit`). Такой механизм позволяет динамически увеличивать размер стека лишь по мере необходимости.

Обращаясь к `CreateThread()`, можно обнулить значение параметра `cbStack`. В этом случае функция создает стек для нового потока, используя аргумент `commit`, внедренный компоновщиком в `exe`-файл. Объем резервируемого пространства всегда равен 1 Мб. Это ограничение позволяет прекращать деятельность функций с бесконечной рекурсией.

Параметр `IpStartAddr` определяет адрес функции потока, с которой должен будет начать работу создаваемый поток. Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и той же стартовой функции.

Параметр `IpvThreadParm` позволяет передавать функции потока какое-либо инициализирующее значение. Оно может представлять собой или просто 32-битное значение, или 32-битный указатель на структуру данных с дополнительной информацией.

Параметр `fdwCreate` определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или `CREATE_SUSPENDED`. В последнем случае система создает поток, затем — его стек, инициализирует элементы соответствующей структуры `CONTEXT` и, приготовившись к исполнению первой команды из функции потока, «придерживает» поток до последующих указаний.

Сразу после возврата из `CreateThread()` и пока еще выполняется вызвавший ее поток, выполняется и новый поток, если только при его создании не указан флаг `CREATE_SUSPENDED`.

Последний *параметр* `lpIDThread` функции `CreateThread()` – это адрес переменной типа `DWORD`, в которой функция вернет идентификатор, приписанный системой новому потоку.

Как и процесс, поток можно завершить тремя способами:

1. Поток самоуничтожается вызовом `ExitThread()` (самый распространенный способ).
 2. Один из потоков данного или стороннего процесса вызывает `TerminateThread()` (этого надо избегать).
 3. Завершается процесс, содержащий данный поток.
- Рассмотрим эти функции.

```
VOID ExitThread(UINT fuExitCode);
```

Функция завершает поток. У функции нет возвращаемого значения, ведь после ее вызова поток перестает существовать. В параметр `fuExitCode` она помещает код завершения потока.

Этот способ завершения потока применяют чаще потому, что после передачи управления от функции потока внутрисистемной функции `StartOfThread` вызывается именно `ExitThread()`, которой передается значение, возвращенное функцией потока.

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Функция завершает поток, идентифицируемый параметром `hThread`, и помещает код завершения в `dwExitCode`. Ее используют в крайнем случае, когда управление потоком потеряно и он ни на что не реагирует.

При завершении потока:

- 1) освобождаются все описатели объектов `User`, принадлежавших потоку. В Win32 большинство объектов принадлежат процессу, содержащему поток, из которого они были созданы. Однако поток может владеть двумя объектами `User`: окнами и ловушками (`hooks`). Когда поток, создавший такие объекты, завершается, система уничтожает их автоматически. Прочие объекты разрушаются, только когда завершается владевший ими процесс;
- 2) объект ядра «поток» переводится в свободное состояние;
- 3) код завершения потока меняется со `STILL_ACTIVE` на код, переданный в функцию `ExitThread()` или `TerminateThread()`;
- 4) если данный поток – последний активный поток в процессе, завершается и сам процесс;
- 5) счетчик числа пользователей объекта ядра «поток» уменьшается на 1.

При завершении потока сопоставленный с ним объект ядра «поток» не освобождается, пока не будут закрыты все внешние ссылки на этот объект.

Когда поток завершился, толку от его описателя другим потокам в системе немного. Единственное, что они могут сделать – вызвать `GetExitCodeThread()` и проверить, завершен ли поток, идентифицируемый описателем `hThread`, и если да, то определить его код завершения.

В примере 3.1 демонстрируется создание многопоточного приложения с помощью функций WinAPI. Основной поток генерирует двумерный массив фиксированного размера, и затем создаются потоки для нахождения суммы его элементов построчно.

Пример 3.1. Листинг многопоточной программы, использующей функции WinAPI

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
//Глобальные переменные, к ним имеют доступ все потоки
const int n = 5; //размерность массива
int a[n][n], b[n];
//Код функции потока
//DWORD WinAPI – тип возвращаемого из функции значения,
//WinAPI указывает, что описана функция потока
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    int num, j;
    //Помещение переданных потоку данных в переменные потока
    //в num будет храниться порядковый номер потока
    num = *((int *)pvParam);
    printf("thread %d: start!\n", num);
    for(j=0; j<n; j++)
        b[num] += a[num][j]; //вычисление суммы по строке
    //Объявление и инициализация переменной, которая будет
    //возвращена в основной поток
    DWORD dwResult_th = num;
    return dwResult_th;
}
//Код основного потока
int main(int argc, char** argv) {
    int x[n]; //с помощью этого массива потоки узнают свои номера
    //Идентификатор потока, определяемый системой,
    //его возвратит функция создания потока
    DWORD dwThreadId[n];
    //Переменная, в которую функция ожидания завершения потоков
    //вернет код его завершения
    DWORD dw;
    //Переменная, в которую будет считано значение,
    //возвращаемое дочерним потоком
    DWORD dwResult_main[n];
    HANDLE hThread[n]; //массив описателей объектов ядра (потоков)
    int i, j, sum;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            a[i][j] = ((i+1)*(j+1))%13;
```

Пример 3.1 (окончание). Листинг многопоточной программы, использующей функции WinAPI

```
//Создание потоков для обработки строк массива
for (i=0;i<n;i++){
    x[i] = i;
    //Выделенный параметр содержит адрес, передаваемый потоку
    //в качестве данных,
    //в нашем случае - порядковый номер его создания
    hThread[i]=CreateThread(NULL, 0, ThreadFunc, (PVOID)&x[i],
                           0, &dwThreadId[i]);

    if(!hThread[i])
        printf("main process: thread %d not execute!",i);
}
//Приостановка выполнения основного потока
//до получения сигнала о завершении всех дочерних потоков,
dw = WaitForMultipleObjects(n,hThread,TRUE,INFINITE);
//Определение значений, переданных завершенными потоками
//и вывод сообщения о завершении дочерних потоков
for (i=0;i<n;i++) {
    GetExitCodeThread(hThread[i],&dwResult_main[i]);
    printf(«\nthread %d is ended\n", (int)dwResult_main[i]);
}
// расчет и вывод результатов
for(i=0;i<n;i++)
    sum += b[i];
printf("\nsum = %d", sum);
return 0;
}
```

3.4. Синхронизация потоков в пользовательском режиме

Система лучше всего работает, когда все потоки могут заниматься своим делом, не взаимодействуя друг с другом. Однако такая ситуация возникает очень редко. Обычно потоки вынуждены синхронизировать друг с другом свои действия. Как уже подробно обсуждалось в предыдущих главах, потоки взаимодействуют в двух основных случаях:

- 1) для корректного использования разделяемого ресурса (взаимное исключение);
- 2) для ожидания потоком завершения какого-либо действия другого потока (условное ожидание).

В ОС Windows реализован ряд механизмов, поддерживающих синхронизацию потоков. Некоторые из них применимы только к потокам одного

процесса, другие универсальны и позволяют синхронизировать любые потоки, существующие в системе.

Здесь и далее под *синхронизацией потоков в пользовательском режиме* мы будем понимать случай, когда взаимодействуют потоки одного процесса, т. е. потоки, находящиеся в общем адресном пространстве.

Interlocked-функции

Большая часть синхронизации потоков связана с атомарным доступом (atomic access) – монопольным захватом ресурса обращающимся к нему потоком.

Interlocked-функции рассматривают в качестве такого ресурса обычную переменную. Как правило, при использовании функций этого вида поток атомарно может изменить значение некоторой глобальной (видимой всем потокам) переменной.

Рассмотрим некоторые атомарные функции, предоставляемые WinAPI.

```
LONG InterlockedExchangeAdd(LONG* plAddend; LONG lIncrement);
```

Эта функция берет переменную, находящуюся по адресу plAddend, и добавляет к ее значению lIncrement. Возвращает функция исходное значение изменяемой переменной. Функция может выполнить и атомарное вычитание – достаточно передать в lIncrement отрицательное число.

Сравним коды потоков из примеров 3.2 и 3.3. Предполагается, что после выполнения потоков значение переменной *x* должно быть равно двум. На самом деле после выполнения обоих потоков из примера 3.2 значение переменной *x* не детерминировано, может стать равным двум или одному. Результат явно нежелательный. В примере 3.3 неатомарное увеличение глобальной переменной *x* заменено соответствующей Interlocked-функцией, в результате можно быть полностью уверенным, что значение переменной *x* после выполнения обоих потоков равно двум.

Пример 3.2. Результат выполнения потоков не детерминирован

```
//Глобальная переменная
long x = 0;
DWORD WINAPI ThreadFunc1(
    PVOID Par)
{
    x++; return 0;
}
DWORD WINAPI ThreadFunc2(
    PVOID
    Par){
    x++; return 0;
}
```

Пример 3.3. Результат выполнения потоков детерминирован (x=2)

```
//Глобальная переменная
long x = 0;
DWORD WINAPI ThreadFunc1(
    PVOID Par)
{
    InterlockedExchangeAdd(&x,1);
    return 0;
}
DWORD WINAPI ThreadFunc2(
    PVOID Par){
    InterlockedExchangeAdd(&x,1);
    return 0;
}
```


Рассмотрим также еще несколько атомарных функций.

```
LONG InterlockedExchange(LONG* pIAddend, LONG lValue);
```

```
PVOID InterlockedExchangePointer(PVOID* ppvTarget, PVOID pvValue);
```

Обе функции атомарно заменяют текущее значение переменной типа LONG, адрес которой передается в первом параметре, на значение, передаваемое во втором. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядной программе первая оперирует с 32-разрядными значениями, а вторая – с 64-разрядными. Все функции возвращают исходное значение переменной.

```
LONG InterlockedCompareExchange(LONG* pIExchange, LONG lExchange, LONG lComparand);
```

```
PVOID InterlockCompareExchangePointer(PVOID* ppvDestination, PVOID pvExchange, PVOID lComparand);
```

Обе функции сравнивают текущее значение переменной, на которую указывает первый параметр, со значением третьего параметра. Если они равны, то первый параметр *pIExchange (*ppvExchange) получает значение lExchange (*pvExchange), в ином случае *pIExchange (*ppvExchange) остается без изменений. В скобках перед именем переменной указаны **, поскольку имеет место указатель на указатель. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядной программе первая оперирует с 32-разрядными значениями, а вторая – с 64-разрядными. Все функции возвращают исходное значение переменной.

Пример 3.4 иллюстрирует использование Interlocked-функции для реализации парадигмы «портфель задач» (см. гл. 2). «Задачей» является обработка строки двумерного массива. В качестве «портфеля задач» используется переменная current, хранящая номер строки, которую необходимо обрабатывать следующей. Interlocked-функция InterlockExchangeAdd() позволяет безопасно использовать «портфель задач» несколькими потоками.

Критические секции

Для управления исключительным доступом к *разделяемым* несколькими потоками данным в WinAPI предусмотрена специальная структура CRITICAL_SECTION. При этом системой не предполагается обращение к полям этой структуры напрямую – протоколы входа и выхода из критической секции реализованы на уровне конкретных функций.

Обычно структуры CRITICAL_SECTION создаются как глобальные переменные, доступные всем потокам процесса. Но ничто не мешает создавать их как локальные переменные или переменные, динамически размещаемые в куче.

Пример 3.4. Многопоточная реализация обработки двумерного массива по строкам с помощью «портфеля задач» с использованием функций WinAPI

```

#include <stdio.h>
#include <conio.h>
#include <windows.h>
const int n = 100; //размерность обрабатываемого массива
const int m = 3;   //количество потоков для обработки
long int a[n][n], current = 0; //портфель задач
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    int i, num, count = 0;
    num = *((int *)pvParam); //определение номера потока
    printf("thread %d: start!\n", num);
    while(count < n) {
        //Неделимым образом узнать номер задачи для выполнения
        //и увеличить портфель задач на единицу
        count = InterlockedExchangeAdd(&current, 1);
        if(count < n) //если задача существует
            for(i=0; i < n; i++) //выполнить задачу
                if(a[count][i] == 1) {a[count][i] = 0;}
    }
    return (DWORD)0;
}
int main(int argc, char **argv) {
    int i, j, x[m];
    DWORD dwThreadId[m];
    HANDLE hThread[m];
    for (i=0; i < n; i++) //заполнение массива для обработки
        for (j=0; j < n; j++)
            a[i][j] = (i*j)%m;
    //Запуск m потоков, номер потока передается как параметр
    for (i=0; i < m; i++) {
        x[i] = i;
        hThread[i] = CreateThread(NULL, 0, ThreadFunc, (PVOID)&x[i],
                                0, &dwThreadId[i]);

        if(!hThread)
            printf("main process: thread %d not execute!", i);
    }
    // ожидание завершения всех потоков
    WaitForMultipleObjects(m, hThread, TRUE, INFINITE);
    for (i=0; i < n; i++) { //вывод на печать обработанного массива
        for (j=0; j < n; j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }
    return 0;
}

```

Есть только два условия, которые необходимо соблюдать. Во-первых, все потоки, которым может понадобиться ресурс, должны знать адрес структуры `CRITICAL_SECTION`, которая защищает этот ресурс. Во-вторых, элементы структуры следует инициализировать до обращения какого-либо потока к защищенному ресурсу.

Рассмотрим функции работы со структурой `CRITICAL_SECTION`.

```
VOID InitializeCriticalSection(CRITICAL_SECTION *pcs);
```

Функция инициализирует переменные структуры `CRITICAL_SECTION`, на которую указывает параметр `pcs`.

```
VOID DeleteCriticalSection(CRITICAL_SECTION *pcs);
```

Функция удаляет структуру `CRITICAL_SECTION`.

При выполнении последней функции сбрасываются все переменные-члены внутри структуры `CRITICAL_SECTION`, т. е. обнуляются или устанавливаются значения по умолчанию. Разумеется, нельзя удалять секцию в тот момент, когда ею пользуется какой-либо поток. Но соблюдение этого правила ложится непосредственно на совесть разработчика приложения.

```
VOID EnterCriticalSection(CRITICAL_SECTION *pcs);
```

Функция входа в критическую секцию. Участок кода, работающий с разделяемым ресурсом, предваряется вызовом этой функции. Функция проверяет значения элементов структуры `CRITICAL_SECTION`, которые, в частности, содержат информацию о том, какой поток пользуется ресурсом.

Функция `EnterCriticalSection()` выполняет следующие действия:

- если ресурс свободен, `EnterCriticalSection()` изменяет элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего возвращает управление вызвавшему ее потоку, который продолжает свою работу, получив доступ к ресурсу;
- если ресурс занят этим же самым потоком, `EnterCriticalSection()` обновляет элементы структуры, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и немедленно возвращает управление;
- если ресурс занят другим потоком, `EnterCriticalSection()` переводит вызвавший ее поток в режим ожидания. Система запоминает, что данный поток хочет получить доступ к ресурсу, и, как только поток, занимавший этот ресурс, освобождает его, вновь начинает выделять ожидающему потоку процессорное время. При этом он получает доступ к ресурсу.

Все эти действия функция выполняет на уровне атомарного (неделимого) доступа.

```
BOOL TryEnterCriticalSection(CRITICAL_SECTION *pcs);
```

Функция пытается войти в критическую секцию. Если ей это не удастся, поток продолжает выполнение без задержки. Функция возвращает значение TRUE, если поток вошел в критическую секцию или уже находится в ней, и FALSE, если попытка была неудачна.

```
VOID LeaveCriticalSection(CRITICAL_SECTION *pcs);
```

Этот вызов должен стоять в конце участка кода, использующего разделяемый ресурс. Функция просматривает элементы структуры CRITICAL_SECTION и уменьшает счетчик числа захватов ресурса вызывающим потоком на 1. Если его значение больше 0, то функция ничего не делает и немедленно возвращает управление вызвавшему ее потоку. Если значение счетчика достигло 0, LeaveCriticalSection() выясняет, есть ли в системе другие потоки, ждущие данный ресурс в вызове EnterCriticalSection(). Если есть хотя бы один такой поток, функция настраивает соответствующим образом элементы структуры CRITICAL_SECTION и отдает ресурс ожидающему потоку. Если же ресурс никому не нужен, LeaveCriticalSection() сбрасывает элементы структуры.

Как и EnterCriticalSection(), функция LeaveCriticalSection() выполняет свои действия на уровне атомарного доступа. Однако LeaveCriticalSection() никогда не приостанавливает поток, а управление возвращает немедленно.

В качестве примера работы с критическими секциями рассмотрим программный код (пример 3.5), содержащий два критических участка. Первый из них очевиден – это работа с разделяемой переменной x. Вторая критическая секция связана с выводом на экран с помощью функции printf(). Дело в том, что printf() не является атомарной функцией, поэтому, если не предпринять специальных усилий, то хотя вывод одного потока будет происходить в определенном программой порядке, но выводы от разных потоков будут перепутаны между собой. Для предотвращения этой ситуации вывод на экран в примере 3.5 также защищен критической секцией.

Пример 3.5. Использование структуры CRITICAL_SECTION для защиты разделяемых несколькими потоками данных

```
#include <stdio.h>
#include <windows.h>
const int m = 2; //количество потоков обработки
LONG x=0; //общий ресурс
CRITICAL_SECTION cs_print, cs_mod; //объявление структур KC
DWORD WINAPI ThreadFunc1(PVOID pvParam){ //код первого потока
    int i;
    //KC: защита вывода на экран
    EnterCriticalSection(&cs_print); //вход в KC
    printf(«Work of the first thread is begining!!! \n»);
```

Пример 3.5 (окончание). Использование структуры `CRITICAL_SECTION` для защиты разделяемых несколькими потоками данных

```

LeaveCriticalSection(&cs_print); //выход из KC
for(i = 0; i<10000; i++){
    //KC: защита изменения разделяемой переменной
    EnterCriticalSection(&cs_mod); //вход в KC
    x++;
    LeaveCriticalSection(&cs_mod); выход из KC
}
return 0;
}

DWORD WINAPI ThreadFunc2(PVOID pvParam){//код второго потока
int i;
//Поскольку в операторе вывода задействованы оба ресурса,
//часть кода находится сразу в двух критических секциях
for(i = 0; i<10; i++){
    EnterCriticalSection(&cs_print);
    EnterCriticalSection(&cs_mod);
    printf("Work of thread two begin!!! x is %d \n", x);
    LeaveCriticalSection(&cs_mod);
    LeaveCriticalSection(&cs_print);
}
return 0;
}

int main(int argc, char** argv){
    int thr_n[m] = {0,1};
    DWORD dwThreadId[m];
    HANDLE hThread[m];
    //Инициализация критических секций:
    //защита вывода на экран
    InitializeCriticalSection(&cs_print);
    //защита изменения разделяемой переменной
    InitializeCriticalSection(&cs_mod);
    hThread[0]=CreateThread(NULL,0,ThreadFunc1, (PVOID) &thr_n[0],
                           0, &dwThreadId[0] );

    if (!hThread)
        printf("main process: thread %d not execute!",0);
    hThread[1]=CreateThread(NULL,0,ThreadFunc2, (PVOID) &thr_n[1],
                           0, &dwThreadId[1]);

    if (!hThread)
        printf("main process: thread %d not execute!",1);
    //Ожидание завершения потоков
    WaitForMultipleObjects(m,hThread,TRUE,INFINITE);
    //Удаление критических секций
    DeleteCriticalSection(&cs_print);
    DeleteCriticalSection(&cs_mod);
    return 0;
}

```

Следует отметить, что поскольку язык Си изначально не снабжен средствами многопоточности, то все стандартные функции (ввода-вывода, строковые и символьные, математические, времени, даты и локализации, динамического распределения памяти, работы с очередями, служебные) неатомарны.

Их незащищенный вызов в многопоточном приложении может приводить к непредвиденным эффектам. Следует аккуратно и корректно использовать такие функции, если, например, их аргументы разделяются несколькими потоками, вкладывать вызов в критическую секцию.

3.5. Синхронизация потоков с помощью объектов ядра

Наиболее общий механизм синхронизации потоков в WinAPI основан на использовании специальных объектов ядра: мьютексов (mutexes), семафоров (semaphores) и событий (events). В отличие от синхронизации в пользовательском режиме с помощью объектов ядра можно синхронизировать потоки, принадлежащие разным процессам.

Ниже рассмотрены общие свойства объектов ядра, реализующих синхронизацию потоков, и конкретные примеры использования объектов для этой цели.

Wait-функции

Почти каждый объект ядра Windows может пребывать в одном из двух состояний – свободном (signaled state) или занятом (unsignaled state). Это характерно, например, для процессов, потоков, заданий, файлов, консольного ввода, уведомления об изменении файлов, событий, ожидаемых таймеров, семафоров, мьютексов.

Переход из одного состояния в другое осуществляется по особым правилам, определенным разработчиками для каждого типа объектов ядра. Например, объект типа «процесс» или «поток» сразу после создания переходит в занятое состояние и становится свободным, когда сопоставленное с ним исполнение команд завершается. Таким образом, по переходу объекта «поток» или «процесс» в свободное состояние можно определить, что поток или процесс завершен.

Потоки могут «засыпать» и в таком состоянии ждать освобождения какого-либо объекта. Для этого используется семейство wait-функций.

Wait-функции позволяют потоку в любой момент приостановиться и ожидать освобождения какого-либо объекта ядра.

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwMilliseconds);
```

Функция ожидает освобождения одного объекта ядра, который идентифицирован описателем, передаваемым в первом параметре. Вторым параметром указывает время в миллисекундах, которое будет ждать поток. Вместо второго параметра можно передать слово `INFINITE`, что будет означать, что поток готов ждать «вечно». Например, по команде

```
WaitForSingleObject(hProc, INFINITE);
```

поток будет «вечно» ждать, пока не завершится процесс, на который указывает описатель `hProc`.

`Wait`-функция возвращает значение типа `DWORD`, проанализировав которое, можно определить, чем конкретно закончилось ожидание.

Код примера 3.6 позволяет выяснить результат срабатывания `Wait`-функции. Используемая `Wait`-функция сообщает системе, что вызывающий поток не должен получать процессорное время до тех пор, пока не завершится указанный процесс или не пройдет 5000 мс. Возвращаемое значение функции указывает, почему поток снова стал планируемым. Если функция возвращает `WAIT_OBJECT_0`, ожидаемый объект свободен, если `WAIT_TIMEOUT` – заданное время ожидания истекло. Относительно другого объекта ядра `Wait`-функция может возвращать и другие значения.

```
DWORD WaitForMultipleObjects(DWORD dwCount, HANDLE* phObjects,
                               BOOL fWaitAll, DWORD dwMilliseconds);
```

Пример 3.6. Фрагмент кода обработки результата срабатывания `Wait`-функции

```
DWORD dw = WaitForSingleObject(hProc, 5000);
switch(dw){
    case WAIT_OBJECT_0: //процесс успешно завершился
        ... //код обработки
        break;
    case WAIT_TIMEOUT: //процесс не завершился за 5000 мс,
                      //ожидание прервано
        ... //код обработки
        break;
    case WAIT_FAILED: //неправильный вызов функции
        ... // код обработки
        break;
}
```

Функция позволяет ожидать освобождения сразу нескольких или одного из списка объектов. Первый параметр `dwCount` определяет количество интересующих объектов ядра. Параметр `phObjects` – это указатель на массив описателей ядра. Параметр `fWaitAll` указывает, следует ли ждать освобождения всех объектов (`TRUE`) или хотя бы одного (`FALSE`).

Важным и очень удобным является то, что обе функции осуществляют свои операции атомарно.

Побочным эффектом успешного ожидания может являться изменение состояния объекта. Это определяется разработчиками особо для каждого из типов объектов ядра. Именно на побочных эффектах основано большинство приемов синхронизации потоков с помощью объектов ядра.

События

Объект «событие» является самой примитивной разновидностью среди объектов ядра, используемых для синхронизации потоков. Он описывается счетчиком числа пользователей и двумя булевыми переменными: тип события (с автосбросом – без автосброса) и его состояние (свободно – занято).

Как правило, события используются для уведомления об окончании какой-либо операции. Объекты ядра «событие» бывают двух видов: со сбросом вручную (`manual-reset event`) и с автосбросом (`auto-reset event`). Первые позволяют возобновить выполнение сразу нескольких ожидающих потоков, вторые – только одного.

Рассмотрим функции WinAPI, работающие с объектом ядра «событие».

```
HANDLE CreateEvent(PSECURITY_ATTRIBUTES psa, BOOL fManualReset,  
                  BOOL fInitialState, PCTSTR pszName);
```

Функция создает объект ядра «событие».

Параметр `psa` указывает на настройки безопасности (как правило, здесь передается `NULL`, выше эти настройки обсуждались более подробно).

Параметр `fManualReset` определяет, будет ли событие со сбросом вручную (`TRUE`) или с автосбросом (`FALSE`).

Параметр `fInitialState` определяет начальное состояние события – свободное (`TRUE`) или занятое (`FALSE`).

Параметр `pszName` позволяет задать событию какое-нибудь имя. Имя позволяет другим процессам получить описатель данного события. Для этого они могут либо использовать `CreateEvent()` с таким же параметром `pszName`, либо открыть событие функцией `OpenEvent()`.

```
HANDLE OpenEvent(DWORD fdwAccess, BOOL fInherit, PCTSTR pszName);
```

Функция открывает объект ядра «событие». Последний параметр, `pszName`, определяет имя объекта ядра. Он всегда должен содержать адрес строки с нулевым символом в конце (передавать `NULL` нельзя). Функции `OpenHandle()` просматривают единое пространство имен объектов ядра,

пытаясь найти совпадение. Если объекта ядра «событие» с указанным именем нет или он другого типа, `Open`-функции возвращают `NULL`. Но если объект ядра «событие» с заданным именем существует, система проверяет, разрешен ли к данному объекту доступ запрошенного (через параметр `fdwAccess`) вида. Если доступ разрешен, таблица описателей в вызывающем процессе обновляется, и счетчик числа пользователей объекта возрастает на единицу. Если присвоить параметру `fInherit` значение `TRUE`, то будет получен наследуемый описатель.

```
BOOL CloseHandle (HANDLE hObj);
```

Функция закрывает описатель `hObj` объекта ядра «событие».

```
BOOL SetEvent (HANDLE hEvent);
```

Функция переводит событие `hEvent` в свободное состояние. Если событие уже было свободно, то никаких действий функция не производит.

```
BOOL ResetEvent (HANDLE hEvent);
```

Функция переводит событие `hEvent` в занятое состояние.

Важным отличием событий с *автосбросом* и *ручным сбросом* является их реакция на вызванную по отношению к ним `Wait`-функцию.

Для событий с автосбросом действует следующее правило. При успешном завершении ожидания освобождения объекта «событие» потоком этот объект автоматически сбрасывается в *занятое* состояние. Для такого объекта обычно не требуется вызывать `ResetEvent()`, так как система сама восстанавливает его состояние. Таким образом, событие с автосбросом можно представить в виде двоичного семафора, где `Wait`-функции или `ResetEvent()` реализуют функцию `P()`, а `SetEvent()` – функцию `V()`.

Для событий с ручным сбросом никаких побочных эффектов не предусмотрено, поэтому только `ResetEvent()` реализуют функцию `P()`. Крупномодульное представление `Wait`-функций можно представить в нашей нотации следующим оператором ожидания:

```
<await (Event>0)>;
```

Поскольку `Wait`-функция не занимает событие, освобождение события с ручным сбросом могут дожидаться сразу несколько потоков.

Рис. 3.2 демонстрирует аналогию между нотацией семафоров, введенной в гл. 2, и объектом ядра «событие».

Ниже приведены два примера использования события – с ручным сбросом и автосбросом соответственно.

В примере 3.7 поток `ThreadOne()` готовит набор данных. Пока происходит подготовка, потоки `ThreadSecond()` и `ThreadThird()` ожидают ее завершения. Как только первый поток завершает формирование данных, он освобождает событие `hEvent`. Поскольку событие инициализировано

с ручным автосбросом, то после освобождения события оба ожидавших его потока могут начать обработку данных.

СОБЫТИЕ		
с автосбросом	с ручным сбросом	нотация
<code>HANDLE hEvent; hEvent = CreateEvent(NULL, FALSE, FALSE, «Event»);</code>	<code>HANDLE hEvent; hEvent = CreateEvent(NULL, TRUE, FALSE, «Event»);</code>	<code>sem Event=0;</code>
установить СОБЫТИЕ в значение «свободно»		
<code>SetEvent(Event);</code>		<code>V(Event): <Event=0;></code>
установить СОБЫТИЕ в значение «занято»		
<code>ResetEvent(Event); WaitForSingleObject(Event, INFINITE);</code>	<code>ResetEvent(Event);</code>	<code>P(Event): <await (Event>0) Event--;></code>
дождаться освобождения СОБЫТИЯ, не занимая его		
нет	<code>WaitForSingleObject(Event, INFINITE);</code>	<code><await (Event>0);></code>

Рис. 3.2. Объект ядра «событие» и двоичный семафор

Пример 3.7. Фрагмент кода, демонстрирующий использование события с ручным сбросом

```

HANDLE hEvent;
DWORD WINAPI ThreadOne(PVOID pvParam) {
    ... //блок подготовки данных
    SetEvent(hEvent);
    return 0;
}
DWORD WINAPI ThreadSecond(PVOID pvParam) {
    WaitForSingleObject(hEvent, INFINITE);
    ... //блок обработки данных, подготовленных потоком ThreadOne
    return 0;
}
DWORD WINAPI ThreadThird(PVOID pvParam) {
    WaitForSingleObject(hEvent, INFINITE);
    ... //блок обработки данных, подготовленных потоком ThreadOne
    return 0;
}
int main(int argc, char** argv) {
    hEvent = CreateEvent(NULL, TRUE, FALSE, "EventBeginExecute");
    HANDLE hThread[3];
    ... //создание потоков ThreadOne, ThreadTwo и ThreadThird
    ... //ожидание завершения работы потоков
    CloseHandle(hEvent);
    return 0;
}

```

В примере 3.8 подобным образом используется событие с автосбросом. Обратите внимание на выделенные изменения в коде. Поскольку первый поток освобождает событие с автосбросом, то блок данных, им подготовленный, сначала обработает один из потоков ThreadSecond() и ThreadThird(), а затем другой (при этом порядок обработки не определен).

Пример 3.8. Фрагмент кода, демонстрирующий использование события с автосбросом

```
HANDLE hEvent;
DWORD WINAPI ThreadOne(PVOID pvParam) {
    ... //блок подготовки данных
    SetEvent(hEvent);
    return 0;
}
DWORD WINAPI ThreadSecond(PVOID pvParam) {
    WaitForSingleObject(hEvent, INFINITE);
    ...//блок обработки данных, подготовленных потоком ThreadOne
    SetEvent(hEvent);
    return 0;
}
DWORD WINAPI ThreadThird(PVOID pvParam) {
    WaitForSingleObject(hEvent, INFINITE);
    ... //блок обработки данных, подготовленных потоком ThreadOne
    SetEvent(hEvent);
    return 0;
}
int main(int argc, char** argv) {
    hEvent = CreateEvent(NULL, FALSE, FALSE, "EventBeginExecute");
    HANDLE hThread[3];
    ... //создание потоков ThreadOne, ThreadTwo и ThreadThird
    ... //ожидание завершения работы потоков
    CloseHandle(hEvent);
    return 0;
}
```

Семафоры

Объект ядра «семафор» используется для учета ресурсов. Семафор представлен счетчиком числа пользователей и двумя значениями: максимальным числом ресурсов, которое может захватить семафор, и счетчиком текущего числа свободных ресурсов.

Семафоры ОС Windows практически реализуют поведение классических семафоров, описанных в гл. 2. Семафор свободен, если счетчик теку-

щего числа свободных ресурсов больше 0, и занят, если счетчик равен 0. Система не допускает присвоения отрицательных значений счетчику текущего числа свободных ресурсов и превышения им максимального числа ресурсов.

Рассмотрим функции, определенные для семафоров.

```
HANDLE CreateSemaphore(PSECURITY_ATTRIBUTES psa,  
                        LONG lInitialCount, BOOL lMaximumCount,  
                        PCTSTR pszName);
```

Функция создает объект ядра «семафор». Параметры `psa` и `pszName` аналогичны описанным для событий. Параметр `lMaximumCount` показывает максимально возможное число ресурсов, обрабатываемых семафором. Параметр `lInitialCount` устанавливает начальное значение счетчика текущего числа свободных ресурсов.

Например, при вызове

```
hSem = CreateSemaphore(NULL, 2, 5, «MySemaphore»);
```

будет создан семафор с пятью ресурсами, причем только два из них доступны изначально.

Любой процесс может получить свой процессозависимый описатель существующего объекта «семафор», вызвав функцию

```
HANDLE OpenSemaphore(DWORD fdwAccess, BOOL fInherit,  
                      PCTSTR pszName);
```

Функция открывает существующий объект ядра «семафор». Параметры функции аналогичны соответствующей функции для события.

Поток получает доступ к ресурсу, вызвав одну из `Wait`-функций с описателем семафора, который соответствует этому ресурсу. `Wait`-функция выполняет проверку значения счетчика текущего числа свободных ресурсов. Если оно больше 0 (семафор свободен), то система атомарно уменьшает значение счетчика на 1, и поток продолжает выполнение. Если же счетчик равен 0, то система переводит вызывающий поток в режим ожидания. Когда другой поток освободит ресурс, система вспомнит об ожидающих потоках – и сделает один из них исполняемым. Таким образом, `Wait`-функции используются в качестве `P()`-функции семафора.

```
BOOL ReleaseSemaphore(HANDLE hSem, LONG lReleaseCount,  
                      LONG* plPreviousCount);
```

Функция освобождает определенное число ресурсов. Первый параметр `hSem` указывает на освобождаемый семафор, второй `lReleaseCount` содержит число освобождаемых ресурсов, а в третий `*plPreviousCount` функция возвращает предыдущее (до вызова функции) значение счетчика текущего числа свободных ресурсов. Таким образом, функция

`ReleaseSemaphore()` реализует несколько расширенную `V()`-функцию семафора.

Пример 3.9 демонстрирует использование семафора для защиты критических секций. В данном фрагменте кода потоки будут по очереди обращаться к области памяти, доступ к которой защищен семафором. Система исключает одновременный доступ двух и более потоков к разделяемому ресурсу.

Пример 3.9. Фрагмент кода, демонстрирующий использование семафора для защиты критических секций

```
HANDLE hSem;
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    while(TRUE) {
        WaitForSingleObject(hSem, INFINITE); //P()
        ... //обработка защищенного блока данных
        ReleaseSemaphore(hSem, 1, NULL); //V()
    }
    return 0;
}
int main(int argc, char** argv){
    hSem = CreateSemaphore(NULL, 1, 1, «ForCS»);
    ...//создание нескольких потоков, исполняющих код ThreadFunc
    CloseHandle(hSem);
    return 0;
}
```

Заметим, что пример 3.9 реализует классическое решение задачи критической секции, но если потоки относятся к одному процессу, то использование структуры `CRITICAL_SECTION` значительно снизит накладные расходы на такую защиту. Если же потоки принадлежат различным процессам, то более логично использование другого объекта ядра – мьютекса.

Мьютексы

Объект ядра «мьютекс» реализует двоичный семафор. Этот объект весьма похож на структуру `CRITICAL_SECTION`, однако с его помощью можно синхронизировать доступ к данным со стороны нескольких процессов.

Рассмотрим функции, определенные для мьютексов.

```
HANDLE CreateMutex(PSECURITY_ATTRIBUTES Ipsa,
                    BOOL fInitialOwner, LPTSTR IpszMutexName);
```

Функция создает объект ядра «мьютекс».

Параметр `Ipsa` указывает на структуру `SECURITY_ATTRIBUTES`.

Параметр `fInitialOwner` определяет, должен ли поток, создающий мьютекс, быть первоначальным владельцем этого объекта. Если он равен `TRUE`, данный поток становится владельцем созданного мьютекса, а объект-мьютекс оказывается в занятом состоянии. Любой другой поток, ожидающий данный мьютекс, будет приостановлен, пока поток, создавший этот объект, не освободит его. Передача `FALSE` в параметре `fInitialOwner` подразумевает, что объект-мьютекс не принадлежит ни одному из потоков и поэтому «рождается свободным». Любой поток, ожидающий освобождения этого объекта, может занять мьютекс, тем самым продолжить свое выполнение.

Параметр `IpszMutexName` содержит либо `NULL`, либо адрес строки (с нулевым символом в конце), идентифицирующей мьютекс. Когда приложение вызывает `CreateMutex()`, система создает объект ядра «мьютекс» и присваивает ему имя, на которое указывает параметр `IpszMutexName`. Это имя используется при совместном доступе к нему нескольких процессов.

`CreateMutex()` возвращает процессорозависимый описатель, определяющий созданный объект-мьютекс.

Разумеется, любой процесс может получить свой процессорозависимый описатель существующего объекта-мьютекса, вызвав функцию

```
HANDLE OpenMutex (DWORD fdwAccess, BOOL fInherit, PCTSTR IpszName);
```

Функция открывает существующий объект ядра «мьютекс».

Параметр `fdwAccess` может быть равен либо `SYNCHRONIZE`, либо `MUTEX_ALL_ACCESS`. Параметр `fInherit` определяет, унаследует ли дочерний процесс описатель данного объекта-мьютекса. Параметр `IpszName` — это имя объекта-мьютекса в виде строки с нулевым символом в конце.

При вызове `OpenMutex()` система сканирует существующие объекты-мьютексы, проверяя, нет ли среди них объекта с именем, указанным в параметре `IpszName`. Обнаружив таковой, она создает описатель объекта, специфичный для данного процесса, и возвращает его вызвавшему потоку. В дальнейшем любой поток из данного процесса может использовать этот описатель при вызове любой функции, требующей такого описателя. Если объекта-мьютекса с указанным именем нет, функция возвращает `NULL`.

Поток занимает объект-мьютекс в результате успешного завершения ожидания одной из `Wait`-функций (например, `WaitForSingleObject()`).

```
BOOL ReleaseMutex (HANDLE hMutex);
```

Функция возвращает мьютекс в свободное состояние. Параметр `hMutex` — это указатель того мьютекса, который освобождается.

Таким образом, критическая секция реализуется с помощью мьютексов по следующей схеме: в начале защищаемого кода ставится `Wait`-функция, а в конце – `ReleaseMutex()`.

Объект-мьютекс отличается от других синхронизирующих объектов ядра тем, что занявшему его потоку передаются права на владение им.

Прочие синхронизирующие объекты могут быть либо свободны, либо заняты – и все. А объекты-мьютексы способны еще и запоминать, какому потоку они принадлежат.

Отказ от объекта-мьютекса происходит, когда ожидавший его поток захватывает этот объект (переводя в занятое состояние), а затем завершается. В таком случае получается, что мьютекс занят и никогда не освободится, поскольку никакой другой поток не сможет этого сделать вызовом `ReleaseMutex()`. Однако система не терпит подобных ситуаций и, заметив, что произошло, автоматически переводит мьютекс в свободное состояние. Поэтому потоки, ожидающие данный объект через `WaitForSingleObject()`, получают возможность захватить его, а упомянутая функция возвращает `WAIT_ABANDONED` вместо `WAIT_OBJECT_0`. Таким образом, поток может узнать, что мьютекс освобожден некорректно.

3.6. Проблемы условной синхронизации

Как уже не раз отмечалось, синхронизация потоков для монопольного доступа к ресурсу – одна из задач, решаемых с помощью семафороподобных объектов. В этом случае объекты ядра «событие», «семафор» и «мьютекс» взаимозаменяемы, поскольку при защите ресурса поток, занявший объект ядра, сам его и освобождает.

Сложнее обстоит дело с решением проблемы условной синхронизации. Правило использования флагов во избежание ситуации «гонок» гласит, что выставляет флаг (переводит объект синхронизации в занятое состояние) один поток, а сбрасывает флаг (переводит объект синхронизации в свободное состояние) – другой. Объекты ядра по-разному реагируют на подобное обращение: события и семафоры допускают, чтобы их занимал один поток и освобождал другой, а мьютексы – нет.

Проиллюстрируем поведение объектов на следующем примере. Основной поток создает объект ядра («событие», «семафор», «мьютекс»), инициализируя его «свободным». Затем основной поток создает три потока. Первый поток занимает объект с помощью `Wait`-функции, второй – пытается освободить объект, а третий – ожидает его освобождения снова `Wait`-функцией. Для того чтобы потоки срабатывали последовательно, используется функция `Sleep()`, которая приостанавливает вызвавший ее по-

ток, тем самым ОС провоцируется на передачу управления другому потоку процесса. Если сброс объекта в свободное состояние возможен, то третий поток должен получить управление штатно, если нет – то этот поток получит управление только после того, как система обнаружит окончание работы первого потока, поэтому он сигнализирует о превышении времени ожидания.

Ниже приведены три программы, реализующие этот эксперимент. Пример 3.10 демонстрирует работу с событиями. Поскольку событие может освобождать поток, который его не занимал, то ожидание третьего потока завершится штатно.

Пример 3.10. Условная синхронизация, реализованная с помощью событий

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
HANDLE hEvent; //описатель событий
//Код потока, занимающего событие первым
DWORD WINAPI First(PVOID pvParam) {
    int num;
    num = *((int *)pvParam);
    DWORD dw = WaitForSingleObject(hEvent, INFINITE);
    switch(dw){ //определение способа завершения ожидания
        //Ожидание успешно завершено, событие занято
        case WAIT_OBJECT_0:
            printf("\n thread %d: OK", num); break;
        //Событие не освобождено, ожидание прервано
        case WAIT_TIMEOUT:
            printf("\n thread %d: Timeout", num); break;
        case WAIT_FAILED: //неправильный вызов функции
            printf("\n thread %d: Failure", num); break;
    }
    Sleep(50000); //задержка, позволяющая отработать всем потокам
    return 0;
}
//Код потока, который пытается освободить событие
DWORD WINAPI Second(PVOID pvParam) {
    int num;
    num = *((int *)pvParam);
    printf("\n thread %d: Start", num);
    //Задержка, позволяющая первому потоку занять событие
    Sleep(5000);
    //освобождение события, занятого в другом потоке
    SetEvent(hEvent);
    printf("\n thread %d: Event free", num);
    return 0;
}
```

Пример 3.10 (окончание). Условная синхронизация, реализованная с помощью событий

```
//Код потока, который пытается освободить событие
DWORD WINAPI Third(PVOID pvParam){
    int num;
    num = *((int *)pvParam);
    //Задержка, позволяющая первому потоку занять событие
    Sleep(10000);
    DWORD dw = WaitForSingleObject(hEvent, INFINITE);
    switch(dw){ //определение способа завершения ожидания
        //Ожидание успешно завершено, событие занято
        case WAIT_OBJECT_0:
            printf("\n thread %d: OK",num); break;
        //Событие не освобождено, ожидание прервано
        case WAIT_TIMEOUT:
            printf("\n thread %d: Timeout",num); break;
        case WAIT_FAILED: //неправильный вызов функции
            printf("\n thread %d: Failure",num); break;
        //Событие занято, но занявший его поток уже завершился
        case WAIT_ABANDONED:
            printf("\n thread %d: OK, but with troubles",num); break;
    }
    return 0;
}

int main(int argc, char** argv){
    int x[3];
    DWORD dwThreadId[3],dw;
    HANDLE hThread[3];
    //Создание события с автосбросом
    hEvent = CreateEvent(NULL,FALSE,TRUE,L"Event1");
    //ВНИМАНИЕ! участок кода, одинаковый для всех трех примеров
    x[0] = 0;
    hThread[0]=CreateThread(NULL,0,First, (PVOID)&x[0],0,
                           &dwThreadId[0]);
    if(!hThread) printf("main: thread %d not execute!",0);
    x[1] = 1;
    hThread[1]=CreateThread(NULL,0,Second, (PVOID)&x[1],0,
                           &dwThreadId[1]);
    if(!hThread) printf("main: thread %d not execute!",1);
    x[2] = 2;
    hThread[2]=CreateThread(NULL,0,Third, (PVOID)&x[2],0,
                           &dwThreadId[2]);
    if(!hThread) printf("main process: thread %d not execute!",1);
    dw = WaitForMultipleObjects(3,hThread,TRUE,INFINITE);
    //Конец одинакового участка кода, в последующих примерах
    //заменен текстом «СОЗДАНИЕ И ОЖИДАНИЕ ЗАВЕРШЕНИЯ ПОТОКОВ»
    // закрытие описателей событий
    CloseHandle(hEvent);
    return 0;
}
```

Пример 3.11 демонстрирует работу с семафорами и по логике идентичен предыдущему. Эта программа также пройдет без ошибок и зависания, поскольку семафор может быть освобожден потоком, который его не занимал.

Пример 3.11. Условная синхронизация, реализованная с помощью семафоров

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
HANDLE hSem; //описатель семафора
//Код потока, занимающего семафор первым
DWORD WINAPI First(PVOID pvParam) {
    int num;
    num = *((int *)pvParam);
    DWORD dw = WaitForSingleObject(hSem, INFINITE);
    switch(dw) { //определение способа завершения ожидания
        //Ожидание успешно завершено, семафор занят
        case WAIT_OBJECT_0:
            printf("\n thread %d: OK", num); break;
        //Семафор не освобожден, ожидание прервано
        case WAIT_TIMEOUT:
            printf("\n thread %d: Timeout", num); break;
        case WAIT_FAILED: //неправильный вызов функции
            printf("\n thread %d: Failure", num); break;
    }
    Sleep(50000); //задержка, позволяющая отработать всем потокам
    return 0;
}
//Код потока, который пытается освободить семафор
DWORD WINAPI Second(PVOID pvParam) {
    int num;
    long prev;
    num = *((int *)pvParam);
    printf("\n thread %d: Start", num);
    //Задержка, позволяющая первому потоку занять семафор
    Sleep(5000);
    //Освобождение семафора, занятого в другом потоке
    ReleaseSemaphore(hSem, 1, &prev);
    printf("\n thread %d: Semaphore released", num);
    return 0;
}
//Код потока, который пытается освободить семафор
DWORD WINAPI Third(PVOID pvParam) {
    int num;
    num = *((int *)pvParam);
```

Пример 3.11 (окончание). Условная синхронизация, реализованная с помощью семафоров

```
//Задержка, позволяющая первому потоку занять семафор
Sleep(10000);
DWORD dw = WaitForSingleObject(hSem, INFINITE);
switch(dw){ //определение способа завершения ожидания
//Ожидание успешно завершено, семафор занят
case WAIT_OBJECT_0:
    printf("\n thread %d: OK",num); break;
//Семафор не освобожден, ожидание прервано
case WAIT_TIMEOUT:
    printf("\n thread %d: Timeout",num); break;
case WAIT_FAILED: //неправильный вызов функции
    printf("\n thread %d: Failure",num); break;
//Семафор занят, но занявший его поток уже завершился
case WAIT_ABANDONED:
    printf("\nthread %d: OK, but with troubles",num);
    break;
}
return 0;
}
int main(int argc, char** argv){
    int x[3];
    DWORD dwThreadId[3],dw;
    HANDLE hThread[3];
    //Создание одноместного семафора с одним свободным местом
    hSem = CreateSemaphore(NULL,1,1,L"SemaphoreStop");
    //СОЗДАНИЕ И ОЖИДАНИЕ ЗАВЕРШЕНИЯ ПОТОКОВ, см. соответствующий
    //участок кода из примера 3.10
    //закрытие описателей событий
    CloseHandle(hSem);
    return 0;
}
```

Пример 3.12 демонстрирует работу с мьютексами и по логике идентичен предыдущим. В этом примере третий поток отработает только после завершения первого потока, который занял мьютекс и закончился, не освободив его. Об этом же сообщит третий поток (см. выделенный код). Если же оператор `Sleep(50000)` в первом потоке заменить на бесконечный цикл, то программа зависнет, причем третий поток никогда не дождется освобождения мьютекса, хотя второй отработает без ошибок и благополучно завершится, не освободив мьютекс, но и не сообщив о возникших трудностях.

Пример 3.12 демонстрирует, что мьютексы нельзя использовать для сигнализации. Кроме того, примеры 3.10–3.12 показывают, что необходимо тщательно анализировать результаты функций, используемых для синхро-

низации, поскольку некоторые действия не вызывают ошибки, но и не изменяют в желательную сторону состояние объекта.

Пример 3.12. Условная синхронизация, реализованная с помощью мьютексов

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
HANDLE hMutex; //описатель мьютекса
//Код потока, занимающего мьютекс первым
DWORD WINAPI First(PVOID pvParam) {
    int num;
    num = *((int *)pvParam);
    DWORD dw = WaitForSingleObject(hMutex, INFINITE);
    switch(dw) { //определение способа завершения ожидания
        //Ожидание успешно завершено, мьютекс занят
        case WAIT_OBJECT_0:
            printf("\n thread %d: OK", num); break;
        //Мьютекс не освобожден, ожидание прервано
        case WAIT_TIMEOUT:
            printf("\n thread %d: Timeout", num); break;
        case WAIT_FAILED: //неправильный вызов функции
            printf("\n thread %d: Failure", num); break;
    }
    Sleep(50000); //задержка, позволяющая отработать всем пото-
кам
    return 0;
}
//Код потока, который пытается освободить мьютекс
DWORD WINAPI Second(PVOID pvParam) {
    int num;
    long prev;
    num = *((int *)pvParam);
    printf("\n thread %d: Start", num);
    //Задержка, позволяющая первому потоку занять семафор
    Sleep(5000);
    //Попытка освобождения мьютекса, занятого в другом потоке,
    //проверка результата
    if (!ReleaseMutex(hMutex))
        printf("\nthread %d: Error in release of mutex", num);
    else printf("\nthread %d: Mutex free", num);
    return 0;
}
//Код потока, который пытается освободить семафор
DWORD WINAPI Third(PVOID pvParam) {
    int num;
    num = *((int *)pvParam);
    //Задержка, позволяющая первому потоку занять мьютекс
    Sleep(10000);
    DWORD dw = WaitForSingleObject(hMutex, INFINITE);
}
```

Пример 3.12 (окончание). Условная синхронизация, реализованная с помощью мьютексов

```
switch(dw){ //определение способа завершения ожидания
//Ожидание успешно завершено, мьютекс занят
case WAIT_OBJECT_0:
    printf("\n thread %d: OK",num); break;
//Мьютекс не освобожден, ожидание прервано
case WAIT_TIMEOUT:
    printf("\n thread %d: Timeout",num); break;
case WAIT_FAILED: //неправильный вызов функции
    printf("\n thread %d: Failure",num); break;
//Мьютекс занят, но занявший его поток уже завершился
case WAIT_ABANDONED:
    printf("\nthread %d: OK, but with troubles",num);
    break;
}
return 0;
}
int main(int argc, char** argv){
    int x[3];
    DWORD dwThreadId[3],dw;
    HANDLE hThread[3];
    //Создание одноместного мьютекса с одним свободным местом
    hMutex = CreateMutex(NULL,FALSE,L"MutexStop");
    //СОЗДАНИЕ И ОЖИДАНИЕ ЗАВЕРШЕНИЯ ПОТОКОВ,
    //см. соответствующий участок кода из примера 3.10
    //Закрытие описателей событий
    CloseHandle(hMutex);
    return 0;
}
```

В заключение приведем код, реализующий решение задачи о кольцевом буфере с помощью функций WinAPI. Из кода видно, что решение ничем не отличается от примера 2.17, рассмотренного в общей нотации в гл. 2.

Задача о кольцевом буфере. Поток-производитель и поток-потребитель разделяют кольцевой буфер, состоящий из 100 ячеек. Производитель передает сообщение потребителю, помещая его в конец очереди буфера. Потребитель извлекает сообщение из начала очереди буфера. Необходимо создать многопоточное приложение, предотвратить такие ситуации, как изъятие сообщения из пустой очереди или помещение сообщения в полный буфер.

Как уже подробно обсуждалось, задача обладает двумя критическими участками кода. Первая критическая секция связана с операциями чтения-записи нескольких потоков в общий буфер, она реализована с помощью двух мьютексов (hMutexP, hMutexC). Во втором случае проблема синхро-

низации определяется тем, что буфер является конечным, а потому запись должна производиться только в те ячейки, которые являются свободными или уже прочитаны потоками-читателями. Ограниченность буфера реализуется парой n -местных семафоров. Значение первого семафора `hSemEmpty` показывает, сколько ячеек в буфере свободно. Ячейка свободна, когда в нее еще не осуществлялась запись или ячейка была прочитана. Значение второго семафора `hSemFull` показывает, сколько ячеек в буфере занято. Естественно, операция записи не может быть выполнена, пока количество занятых ячеек равно 100 (или количество свободных ячеек равно 0), а операция чтения не может быть выполнена, пока количество свободных ячеек равно 100 (или количество занятых ячеек равно 0). Пример 3.13 демонстрирует законченный код решения задачи о кольцевом буфере с помощью функций WinAPI.

Пример 3.13. Решение задачи о кольцевом буфере с помощью функций WinAPI

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>
//m - потоков-потребителей и k - потоков-производителей
const int n = 100, m = 7, k = 3;
int buf[n], front = 0, rear = 0;
HANDLE hSemFull, hSemEmpty, hMutexP, hMutexC;
//Код потока-производителя, пополняющего буфер
DWORD WINAPI Producer(PVOID pvParam) {
    long prev;
    int num = *((int *)pvParam);
    printf("thread %d (producer): start!\n", num);
    while(true) {
        WaitForSingleObject(hSemEmpty, INFINITE);
        WaitForSingleObject(hMutexP, INFINITE);
        buf[rear] = rand() % n;
        printf("\nproducer %d: data = %d to\n", num, buf[rear], rear);
        rear = (rear + 1) % n;
        Sleep(1000);
        ReleaseMutex(hMutexP);
        ReleaseSemaphore(hSemFull, 1, &prev);
    }
    return 0;
}
//Код потока-потребителя, берущего данные из буфера
DWORD WINAPI Consumer(PVOID pvParam) {
    int num, data;
```


Пример 3.13 (окончание). Решение задачи о кольцевом буфере с помощью функций WinAPI

```

long prev;
num = *((int *)pvParam);
printf("thread %d (consumer): start!\n", num);
while(true){
    WaitForSingleObject(hSemFull, INFINITE);
    WaitForSingleObject(hMutexC, INFINITE);
data = buf[front];
    printf("\nconsumer %d: data = %d from %d", num, data, front);
    front = (front+1)%n;
    Sleep(1000);
    ReleaseMutex(hMutexC);
    ReleaseSemaphore(hSemEmpty, 1, &prev);
}
return 0;
}
int main(int argc, char** argv){
    int i, x[k+m];
    DWORD dwThreadId[k+m];
    HANDLE hThread[k+m];
    hSemEmpty = CreateSemaphore(NULL, n, n, L"Empty");
    hSemFull = CreateSemaphore(NULL, 0, n, L"Full");
    hMutexP = CreateMutex(NULL, FALSE, L"MutexP");
    hMutexC = CreateMutex(NULL, FALSE, L"MutexC");
    for(i=0; i<k; i++){
        x[i] = i;
        hThread[i] = CreateThread(NULL, 0, Producer, (PVOID) &x[i],
                                0, &dwThreadId[i]);

        if(!hThread)
            printf("main process: thread %d not execute!", i);
    }
    for(i=k; i<k+m; i++){
        {
            x[i] = i;
            hThread[i] = CreateThread(NULL, 0, Consumer, (PVOID) &x[i],
                                    0, &dwThreadId[i]);

            if(!hThread)
                printf("main process: thread %d not execute!", i);
        }
    }
    WaitForMultipleObjects(k+m, hThread, TRUE, INFINITE);
    // закрытие описателей событий
    CloseHandle(hSemFull); CloseHandle(hSemEmpty);
    CloseHandle(hMutexP); CloseHandle(hMutexC);
    return 0;
}

```

3.7. Проецируемые в память файлы

Проецируемые файлы позволяют резервировать область оперативной памяти (ОП) и связывать ее с областью внешней памяти, соответствующей некоторому файлу. Как только файл спроецирован в ОП, к нему можно обращаться так, как будто он целиком в нее загружен.

Проецируемые файлы применяются в следующих случаях:

- для загрузки и выполнения `exe`- и `DLL`-файлов, что позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению;
- для доступа к файлу данных, размещенному на диске, что позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого;
- как способ разделения данных между несколькими процессами, выполняемыми на одной машине.

Далее нас будут интересовать объекты ядра «файл» и «проекция файла» именно с точки зрения организации обмена данными между потоками, принадлежащими разным процессам.

Работа с проекцией файла должна предваряться ее созданием и связыванием с реальным файлом на диске, что можно представить следующим набором операций:

- 1) создать или открыть объект ядра «файл», идентифицирующий дисковый файл, который требуется использовать как проецируемый в память;
- 2) создать объект ядра «проекция файла», чтобы сообщить системе размер файла и способ доступа к нему;
- 3) указать системе, как необходимо спроецировать в адресное пространство процесса объект «проекция файла» (целиком или только его часть).

По завершении работы с проекцией файла необходимо корректно уведомить об этом систему, выполнив следующие операции:

- 1) сообщить системе об отмене проецирования на адресное пространство процесса объекта ядра «проекция файла»;
- 2) закрыть объект «проекция файла»;
- 3) закрыть объект ядра «файл».

Рассмотрим функции WinAPI, необходимые для работы с проекциями файлов.

```
HANDLE CreateFile(LPCSTR lpFileName,  
                  DWORD dwDesiredAccess,  
                  DWORD dwShareMode,  
                  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
                  DWORD dwCreationDisposition,  
                  DWORD dwFlagsAndAttributes,  
                  HANDLE hTemplateFile);
```

Функция создает объект ядра «файл» с именем, передаваемым в параметре `lpFileName`, содержащем строку с именем файла на диске. Если файла с указанным именем на диске не существует, то функция его создает.

Параметр `dwDesiredAccess` указывает на способ доступа к содержимому файла. Возможные значения этого параметра описаны в табл. 3.2.

Параметр `dwShareMode` задает тип совместного доступа к данному файлу, возможные значения которого описаны в табл. 3.3. Именно этот параметр отвечает за возможность разделяемого или монопольного доступа к файлу. Если файл уже монопольно «открыт», то функция `CreateFile()` вернет ошибку.

Создав или открыв указанный файл, `CreateFile()` возвращает его дескриптор, в случае неудачи — `INVALID_HANDLE_VALUE`.

Таблица 3.2

Значение	Описание
0	Содержимое файла нельзя считывать или записывать. Это значение указывается, если необходимо лишь получить атрибуты файла
<code>GENERIC_READ</code>	Чтение файла разрешено
<code>GENERIC_WRITE</code>	Запись в файл разрешена
<code>GENERIC_READ GENERIC_WRITE</code>	Разрешены чтение и запись

Вызов `CreateFile()` указывает операционной системе, где находится физическая память проекции файла (file mapping): на диске, в Сети, на приводе CD-ROM или в другом месте. Далее следует создать объект ядра «проекция файла» и сообщить системе, какой объем ОП нужен для проекции.

Таблица 3.3

Значение	Описание
0	Файл не подлежит открытию «со стороны»
<code>FILE_SHARE_READ</code>	Попытка постороннего процесса открыть файл с флагом <code>GENERIC_WRITE</code> не удастся
<code>FILE_SHARE_WRITE</code>	Попытка постороннего процесса открыть файл с флагом <code>GENERIC_READ</code> не удастся
<code>FILE_SHARE_READ FILE_SHARE_WRITE</code>	Посторонний процесс может открывать файл без ограничений

```
HANDLE CreateFileMapping(HANDLE hFile,
                          LPSECURITY_ATTRIBUTES Ipsa,
                          DWORD fdwProtect,
                          DWORD dwMaximumSizeHigh,
                          DWORD dwMaximumSizeLow,
                          LPSTR pszName);
```

Функция создает объект ядра «проекция файла», который, в частности, содержит информацию о требуемом количестве оперативной памяти для проекции файла (при этом сама память не выделяется!) и ее атрибутах защиты.

Таким образом, объект ядра «проекция файла» является некоторым посредником между процессом, ОС и файлом. Для одного и того же файла можно создать несколько проекций (даже в одном потоке). Система допускает по своему усмотрению буферизацию проекций, поэтому необходимо помнить, что если создано несколько проекций одного файла, то может наблюдаться временное несоответствие данных в этих проекциях. Гарантированная синхронизация данных файла и его проекции проводится с помощью специальной функции `FlushViewOfFile()`, описанной ниже. Такая синхронизация, вообще говоря, не обязательна и определяется целями создания проекции файла.

Параметр `hFile` идентифицирует дескриптор файла, который проецируется на адресное пространство процесса. Этот дескриптор получается на предыдущем этапе, при вызове `CreateFile()`. Параметр `Ipsa` – это указатель на структуру `SECURITY_ATTRIBUTES`; обычно для установки защиты по умолчанию ему присваивается `NULL`. Параметр `fdwProtect` определяет желательные атрибуты защиты, наиболее часто используемые приведены в табл. 3.4.

Параметры `dwMaximumSizeHigh` и `dwMaximumSizeLow` являются наиболее важными. Основное назначение `CreateFileMapping()` – убедиться, что для объекта «проекция файла» доступен нужный объем физической памяти. Так как Win32 позволяет работать с файлами, размер которых выражается 64-разрядными числами, в параметре `dwMaximumSizeHigh` указывают старшие 32 бита, а в `dwMaximumSizeLow` – младшие 32 бита этого значения. Для файлов размером 4 Гб и менее параметр `dwMaximumSizeHigh` всегда равен нулю. Для создания объекта «проекция файла», отражающего текущий размер файла, следует передать в обоих параметрах нули.

Таблица 3.4

Атрибут защиты	Описание
<code>PAGE_READONLY</code>	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла. При этом в функции <code>CreateFile()</code> необходимо выставить флаг <code>GENERIC_READ</code>
<code>PAGE_READWRITE</code>	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. При этом в функции <code>CreateFile()</code> необходимо выставить флаг <code>GENERIC_READ GENERIC_WRITE</code>
<code>PAGE_WRITECOPY</code>	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. Запись приведет к созданию закрытой копии страницы. При этом в функции <code>CreateFile</code> необходимо выставить флаг либо <code>GENERIC_READ</code> , либо <code>GENERIC_READ GENERIC_WRITE</code>

Параметр `pszName` — это строка с нулевым байтом в конце. В ней указывается имя данного объекта «проекция файла», которое используется для доступа к объекту другого процесса.

Функция `CreateFileMapping()` создает объект «проекция файла» и возвращает его описатель в вызвавший функцию поток. Если объект создать не удалось, возвращается нулевой описатель (`NULL`).

Когда объект «проекция файла» создан, нужно, чтобы система резервировала область адресного пространства процесса под данные файла и непосредственно отображала их.

```
LPVOID MapViewOfFile(HANDLE hFileMappingObject,
                     DWORD dwDesiredAccess,
                     DWORD dwFileOffsetHigh,
                     DWORD dwFileOffsetLow,
                     DWORD dwNumberOfBytesToMap);
```

Функция выделяет в адресном пространстве процесса запрошенное количество памяти, которая связывается с объектом «проекция файла». Для одной проекции можно выполнить несколько функций `MapViewOfFile()`, как бы связывая различные фрагменты файла с определенными (но различными) областями адресного пространства процесса.

Параметр `hFileMappingObject` идентифицирует описатель объекта «проекция файла», возвращаемый предшествующим вызовом либо функции `CreateFileMapping()`, либо функции `OpenFileMapping()`. Параметр `dwDesiredAccess` идентифицирует вид доступа к данным, возможные варианты которого перечислены в табл. 3.5. Остальные три параметра относятся к резервированию области адресного пространства процесса и к отображению на него физической памяти. При этом не обязательно проецировать на адресное пространство весь файл сразу. Напротив, можно спроецировать лишь малую его часть, которая в таком случае называется *представлением* (view).

Проецируя на адресное пространство процесса представление файла, необходимо сообщить ОС, какой байт файла данных считать в представлении первым. Для этого предназначены параметры `dwFileOffsetHigh` и `dwFileOffsetLow`. Поскольку Win32 поддерживает файлы длиной до 18 Тб, приходится определять смещение в файле как 64-разрядное число: старшие 32 бита передаются в параметре `dwFileOffsetHigh`, а младшие — в параметре `dwFileOffsetLow`. Смещение в файле должно быть четным числом и кратно гранулярности выделения ресурсов в данной системе (в Win32 она составляет 64 Кб). Кроме смещения потребуется указать размер представления, т. е. сколько байт файла данных должно быть спроецировано на адресное пространство. Это равносильно тому, что задан размер

области, резервируемый в адресном пространстве. Размер указывается в параметре `dwNumberOfBytesToMap`, который представляет собой 32-битную переменную, так как размер представления не может превышать 4 Гб. Если в этом параметре задан 0, система попытается спроецировать представление (начиная с указанного смещения и до конца файла).

Таблица 3.5

Значение	Описание
<code>FILE_MAP_WRITE</code>	Файловые данные можно считывать и записывать. При этом в функции <code>CreateFileMapping()</code> необходимо выставить флаг <code>PAGE_READWRITE</code>
<code>FILE_MAP_READ</code>	Файловые данные можно только считывать. При этом в функции <code>CreateFileMapping()</code> необходимо выставить любой из следующих флагов: <code>PAGE_READONLY</code> , <code>PAGE_READWRITE</code> или <code>PAGE_WRITECOPY</code>
<code>FILE_MAP_ALL_ACCESS</code>	То же, что и <code>FILE_MAP_WRITE</code>
<code>FILE_MAP_COPY</code>	Файловые данные можно считывать и записывать. Запись приводит к созданию закрытой копии страницы. При этом в функции <code>CreateFileMapping()</code> необходимо выставить любой из следующих флагов: <code>PAGE_READONLY</code> , <code>PAGE_READWRITE</code> или <code>PAGE_WRITECOPY</code> .

Когда потребность в данных файла, спроецированного на область адресного пространства процесса, отпадет, необходимо освободить область.

```
BOOL UnmapViewOfFile(LPVOID lpBaseAddress);
```

Функция освобождает область адресного пространства процесса. Ее единственный параметр, `lpBaseAddress`, указывает базовый адрес возвращаемой системе области. Он должен совпадать со значением, полученным функцией `MapViewOfFile()`. Вызывать функцию `UnmapViewOfFile()` обязательно. Если это не будет сделано, память не освободится до завершения процесса. Кроме того, повторный вызов `MapViewOfFile` приводит к резервированию новой области в пределах адресного пространства процесса, но ранее выделенная память не освобождается.

Для повышения производительности при работе с представлением файла система буферизует страницы данных в файле и не обновляет немедленно дисковый образ файла (disk image of file). Когда же вызывается `UnmapViewOfFile()`, система вносит все измененные данные, накопленные в памяти, в дисковый образ файла.

При необходимости программист может задать точки синхронизации проекции файла и его «жесткой» копии самостоятельно.

```
BOOL FlushViewOfFile(LPVOID IpBaseAddress,  
                      DWORD dwNumberOfBytesToFlush);
```

Функция заставляет систему записать все измененные данные в дисковый образ файла. Параметр `IpBaseAddress` должен содержать адрес проецируемого представления, возвращенный в предыдущем вызове функции `MapViewOfFile()`. Через параметр `dwNumberOfBytesToFlush` передается количество байт, которые необходимо записать на диск.

Если `FlushViewOfFile()` вызвана при отсутствии измененных данных, она сразу же вернет управление основной программе.

В случае проецируемых файлов, физическая память которых расположена на сетевом диске, `FlushViewOfFile()` гарантирует, что файловые данные будут перекачаны с рабочей станции. Но она не гарантирует, что сервер, обеспечивающий доступ к этому файлу, запишет данные на удаленный диск, так как он может просто кэшировать их. Для подстраховки при создании объекта «проекция файла» и последующем проецировании его представления следует использовать флаг `FILE_FLAG_WRITE_THROUGH`. При открытии файла с этим флагом функция `FlushViewOfFile()` вернет управление только после сохранения на диске сервера всех файловых данных.

У функции `UnmapViewOfFile()` есть одна особенность. Если первоначально представление было спроецировано с флагом `FILE_MAP_COPY`, любые изменения, внесенные в файловые данные, на самом деле производятся над копией этих данных, хранящихся в страничном файле. Вызванной в этом случае функции `UnmapViewOfFile()` нечего обновлять в дисковом файле, и она просто иницирует возврат системе страниц физической памяти. Все изменения в данных на этих страницах теряются. Поэтому о сохранении измененных данных следует заботиться самостоятельно.

Любой открытый объект ядра должен быть закрыт, иначе в процессе начнется утечка ресурсов. Конечно, по завершении процесса ОС автоматически закроет объекты, оставленные открытыми. Однако поскольку долгая работа процесса приводит к накоплению незакрытых описателей, следует придерживаться правил «хорошего тона» и составлять код так, чтобы открытые объекты всегда закрывались, как только они становятся ненужными. Чтобы закрыть объекты «проекция файла» и «файл», нужно дважды вызвать функцию `CloseHandle()` с соответствующими описателями объектов ядра.

3.8. Совместный доступ процессов к данным через механизм проецирования

В Win32 механизм проецирования файлов – это единственный способ использования совместных данных потоками нескольких процессов. Все прочие методы совместного доступа и обмена данными, например основанные на вызове `SendMessage()` или `PostMessage()`, также применяют механизм проецирования файлов.

Совместное использование данных в этом случае происходит следующим образом. Два или более процессов проецируют в память представления одного и того же объекта «проекция файла», т. е. разделяют между собой одни и те же страницы физической памяти. Когда один процесс записывает данные в представление совместного объекта «проекция файла», изменения немедленно отражаются в представлениях других процессов. Ясно, что для гарантии синхронизации все процессы должны использовать одинаковое имя объекта «проекция файла».

Функция `OpenFileMapping()` позволяет потоку открыть уже существующую проекцию файла другого потока, используя ее имя:

```
HANDLE OpenFileMapping(DWORD dwDesiredAccess,  
                        BOOL bInheritHandle, LPSTR lpName);
```

Эта функция аналогична `CreateFileMapping()`, за тем исключением, что предполагает существование объекта «проекция файла»; если такового нет, новый объект не создается.

Параметр `dwDesiredAccess` определяет права доступа (`FILE_MAP_READ`, `FILE_MAP_WRITE`, `FILE_MAP_ALL_ACCESS` или `FILE_MAP_COPY`), а параметр `bInheritHandle` указывает, должны ли порождаемые процессы автоматически наследовать дескриптор данного объекта «проекция файла».

Значение, возвращаемое функцией, – это процессорозависимый дескриптор объекта «проекция файла», созданного первым процессом.

Если `OpenFileMapping()` не найдет объект «проекция файла» с указанным именем, она вернет `NULL`. Если же возвращается допустимый дескриптор, то для проецирования данных в адресное пространство процессу необходимо использовать `MapViewOfFile()`.

Ясно, что для корректной работы один процесс должен сначала создать объект вызовом `CreateFileMapping()`; тогда остальные процессы смогут открыть этот объект, вызвав `OpenFileMapping()`.

Пример 3.14 демонстрирует возможность обмена данными между процессами с помощью разделения ими объекта «проекция файла». Следует запустить несколько копий исполняемого файла примера. Оператор `Sleep(1000)` задержит начало выполнения программы, что позволит

к первому считыванию данных из файла открыть доступ к нему нескольким процессам.

Пример 3.14. Использование объекта ядра «проекция файла» для обмена данными между процессами

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>
int main(int argc, char** argv){
    HANDLE hFile, hFileMap, hMutex;
    int *masFile;
    hMutex = CreateMutex(NULL, FALSE, "MyMutex");
    hFile = Create-
File(L"C:\\Test.dat",GENERIC_READ|GENERIC_WRITE,
        FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,
        OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
                                0, 100, NULL);

    Sleep(1000);
    masFile=(int*)MapViewOfFile(hFileMap,FILE_MAP_WRITE,0,0,100);
    WaitForSingleObject(hMutex,INFINITE);
    printf("\n I read %d and %d", masFile[0], masFile[1]);
    if (masFile[0] == 0){
        masFile[0] = 100; masFile[1] = 200;
        FlushViewOfFile(masFile,sizeof(int)*2);
        printf("\n I write!");
    }
    else printf("\n I read %d and %d",masFile[0],masFile[1]);
    ReleaseMutex(hMutex);
    UnmapViewOfFile(masFile);
    CloseHandle(hFileMap);
    CloseHandle(hFile);
    return 0;
}
```

Каждый процесс выполняет следующую последовательность действий. Читает содержимое файла с именем «C:\\Test.dat» (если файла не существует, то он будет создан). Если файл пуст, то в него будут записаны два значения типа `int`. Если же в файле что-то содержится, то процесс считывает два значения типа `int` и выводит их на экран. Доступ к представлению является критической секцией и защищен мьютексом. Работа с файлом организована через его проекцию в адресное пространство процесса. Если запустить несколько экземпляров программы, то первый процесс, прошедший защиту мьютекса, запишет данные в файл, а остальные будут их считывать и выводить на экран.

Контрольные вопросы и задания

1. Опишите понятие объекта ядра Windows.
2. Каким образом объекты ядра совместно используются несколькими процессами?
3. Опишите общий шаблон и алгоритм работы семейства Create-функций Win32 API. В чем состоит отличие функций этого семейства от Open-функций?
4. Опишите общий шаблон и алгоритм работы семейства Open-функций Win32 API. В чем состоит отличие функций этого семейства от Create-функций?
5. Опишите организацию процесса в ОС MS Windows.
6. Опишите основные параметры процесса в ОС MS Windows.
7. Перечислите способы порождения и завершения процесса в ОС MS Windows.
8. Опишите особенности создания и использования потоков в Win32 API.
9. Опишите основные параметры потока в ОС MS Windows.
10. Перечислите способы порождения и завершения потока с использованием функций WinAPI.
11. Расскажите о синхронизации потоков в пользовательском режиме с помощью Interlocked-функций.
12. Расскажите об особенностях организации критической секции с помощью структуры `CRITICAL_SECTION`.
13. Расскажите об особенностях синхронизации потоков с помощью объектов ядра. Какова роль `Wait`-функций в синхронизации потоков?
14. Определите объект ядра «событие». Расскажите об организации исключительного доступа к ресурсу с помощью событий.
15. Расскажите о функциях Win32 API для работы с событием. Поясните особенности реализации условной синхронизации с помощью событий.
16. Определите объект ядра «семафор». Расскажите об организации исключительного доступа к ресурсу с помощью семафоров.
17. Расскажите о функциях Win32 API для работы с семафорами. Поясните особенности реализации условной синхронизации с помощью семафоров.
18. Определите объект ядра «мьютекс». Расскажите об организации исключительного доступа к ресурсу с помощью мьютексов.
19. Расскажите о функциях Win32 API для работы с мьютексами. Сходства и отличия семафоров и мьютексов в Win32 API.
20. Прокомментируйте решение задачи о кольцевом буфере с использованием функций Win32 API из примера 3.13.
21. Для чего в ОС MS Windows используется проекция файла?

22. Опишите алгоритм создания объекта ядра «проекция файла».
23. Расскажите о функциях Win32 API для работы с проекцией файла. Какова процедура синхронизации файла и его проекции?
24. Объясните понятие представления в Win32 API и принципы работы с ним. Могут ли разделять представление несколько потоков одного процесса? несколько потоков разных процессов?
25. Опишите способ организации совместного доступа нескольких процессов к данным через механизм проецирования файлов. Прокомментируйте код примера 3.14.
26. Преобразуйте код примера 3.14 таким образом, чтобы первый прошедший защиту мьютекса процесс записывал в файл строку, а все остальные считывали ее и выводили на экран.
27. Что произойдет при работе нескольких процессов, выполняющих код примера 3.14, если убрать защиту критической секции?

Задачи

Для приведенных ниже задач следует соблюдать следующий порядок работы:

1. Разработать алгоритм решения задачи с учетом разделения вычислений между несколькими потоками.
2. Определить критические области алгоритма.
3. Если в задаче явно не указан механизм синхронизации, то использовать оптимальный с Вашей точки зрения подход.
4. Реализовать алгоритм с применением функций WinAPI.
- 3.1–3.6. Запрограммируйте решение задач 1.1–1.6 с помощью активного ожидания и, если необходимо, синхронизации в пользовательском режиме, используя функции WinAPI.
- 3.7–3.10. Запрограммируйте решение задач 1.3–1.6 с помощью объектов ядра Windows. Какие объекты ядра в решении использованы для реализации взаимного исключения, а какие – для условной синхронизации потоков?
- 3.11–3.16. Запрограммируйте решение задач 2.5–2.10 с помощью объектов ядра Windows. Какие объекты ядра в решении использованы для реализации взаимного исключения, а какие – для условной синхронизации потоков?
- 3.17. *Третья задача о Винни-Пухе, или «Опять неправильные пчелы».* В улье живет N пчел ($N > 3$). Каждая пчела может собирать мед и сторожить улей. Ни одна пчела не покинет улей, если кроме нее в нем нет других пчел. Каждая пчела приносит за раз одну порцию меда. Всего в улей может войти M порций меда. Винни-Пух спит, пока меда в улье меньше половины, но как только его становится достаточно, он просыпается и пытается

достать весь мед из улья. Если в улье находится менее чем три пчелы, Винни-Пух забирает мед, убегает, съедает мед и снова засыпает. Если в улье пчел больше, они кусают Винни-Пуха, он убегает, лечит укус и снова бежит за медом. Пока Винни разоряет улей, ни одна пчела не может прийти на помощь тем, кто оказался внутри улья. Создать многопоточное приложение, моделирующее поведение пчел и медведя.

3.18. *Задача о читателях и писателях.* Базу данных разделяют два типа потоков – читатели и писатели. Читатели выполняют транзакции, которые просматривают записи базы данных, транзакции писателей и просматривают и изменяют записи. Предполагается, что вначале БД находится в непротиворечивом состоянии (т. е. отношения между данными имеют смысл). Каждая отдельная транзакция переводит БД из одного непротиворечивого состояния в другое. Для предотвращения взаимного влияния транзакций поток-писатель должен иметь исключительный доступ к БД. Если к БД не обращается ни один из потоков-писателей, то выполнять транзакции могут одновременно сколько угодно читателей. Создать многопоточное приложение с потоками-писателями и потоками-читателями. Реализовать решение с приоритетом писателей, используя семафоры.

3.19. *Сериализация транзакций базы данных.* Базу данных разделяют четыре типа потоков: читающие данные, изменяющие данные, вставляющие новые данные, удаляющие существующие данные. Читатели выполняют транзакции, которые просматривают записи базы данных, транзакции остальных потоков просматривают и изменяют (причем, возможно, вставляют новые записи или удаляют существующие) базу данных. Предполагается, что вначале БД находится в непротиворечивом состоянии (т. е. отношения между данными имеют смысл). Каждая отдельная транзакция переводит БД из одного непротиворечивого состояния в другое непротиворечивое состояние. Для предотвращения взаимного влияния транзакций потоки-писатели, а также потоки, вставляющие и удаляющие данные, должны иметь исключительный доступ к БД. Если к БД не обращается ни один из потоков, изменяющих, удаляющих или вставляющих данные, то выполнять транзакции могут одновременно сколько угодно читателей. Создать многопоточное приложение с потоками-писателями и потоками-читателями. Реализовать решение с приоритетом читателей.

3.20. *Задача о каннибалах.* Племя из n дикарей ест вместе из большого котла, который вмещает m кусков тушеного миссионера. Когда дикарь хочет обедать, он ест из горшка один кусок, если только горшок не пуст, иначе дикарь будит повара и ждет, пока тот не наполнит горшок. Повар, сварив обед, пробует его и засыпает. Создать многопоточное приложение, моделирующее обед дикарей. При решении задачи пользоваться семафорами.

3.21. *Задача про охоту на носорога.* Как известно, носорог обладает одним недостатком – слабым зрением – и одним достоинством – при его массе и бронировании слабое зрение перестает быть его проблемой. Два охотника – Смит и Джонс – пытаются подстрелить одинокого носорога посреди саванны. Для этого они идут вдоль противоположных сторон огромного прямоугольного участка в противоположных направлениях и выглядывают зверя. Носорог, чуя, что вокруг творится что-то неладное, хаотично перемещается по полю в поисках беспокоящих его людей. Требуется создать многопоточное приложение, моделирующее поведение носорога и охотников. Саванну представить двумерным массивом большой размерности, заполненным нулями и одной единицей (носорог). Охотники реализуются двумя одинаковыми потоками, независимо обходящими массив по противоположным сторонам навстречу друг другу, шагая с разной скоростью (разными задержками), начав с противоположных углов. Охотники целятся перпендикулярно траектории движения внутрь массива. Дойдя до противоположного конца саванны, охотник поворачивает на 180° и продолжает обход в противоположном направлении. После каждого шага охотник оценивает обстановку (задерживается на таймере). Если охотник, сделав шаг и оценив обстановку, оказывается на линии обстрела другого охотника, он в испуге прыгает на два шага вперед. Если охотник, сделав шаг и оценив обстановку, оказывается на одной линии с носорогом, то он получает трофей. Носорог реализуется независимым потоком, т. е. единица перемещается по массиву независимо от процессов-охотников с некоторой задержкой на каждом шаге. Если носорог делает шаг и оказывается на одной линии с охотником, то он, не разбирая дороги, по прямой и без задержек несется на охотника, если он успевает застать охотника в ячейке до того, как тот осмотрелся, то охотник выбывает из игры. Следует учесть, что носорог не умеет прыгать и летать. Охота заканчивается, если либо не останется ни одного охотника, либо один из охотников получит трофей.

3.22. *Военная задача.* Петька и Василий Иванович играют в морской бой. Акватория – это двумерный массив размерностью $N \times N$. Каждый корабль в акватории игрока занимает один элемент и стоит некоторое количество очков. Начальная стоимость флотилии каждого из игроков равна F очков, количество и расположение кораблей в ней неизвестно. Стоимость каждого снаряда S очков. Стрельба ведется по очереди до тех пор, пока у одного из игроков либо не будут уничтожены все корабли, либо стоимость потраченных снарядов не превысит суммарную стоимость всего того, что ими можно уничтожить. Создать многопоточное приложение, моделирующее а) одиночную игру, б) серию из M игр.

3.23. *Задача о супермаркете.* В супермаркете работают три кассы, покупатели заходят в супермаркет, делают покупки и становятся в очередь

к случайному кассиру. Пока очередь пуста, кассир спит; как только появляется покупатель, кассир просыпается. Покупатель спит в очереди, пока не подойдет к кассиру. Создать многопоточное приложение, моделирующее рабочий день супермаркета. Покупатели появляются в супермаркете целый день, группами и поодиночке в случайное время.

3.24. *Задача о больнице.* В больнице два врача-терапевта. Пациенты, придя в больницу, становятся в одну из очередей к этим терапевтам. Каждый врач-терапевт принимает пациента, чья очередь подошла, выслушивает его жалобы и либо выписывает рецепт, либо направляет к невропатологу, или к хирургу, или к эндокринологу. Невропатолог, хирург и эндокринолог также по очереди лечат пациентов. Пациенты стоят в очереди к врачам и никогда их не покидают до приема. Создать многопоточное приложение, моделирующее рабочий день клиники.

3.25. *Первая задача о гостинице.* В гостинице N номеров с ценой X руб., M номеров с ценой Y руб. и K номеров с ценой Z руб. ($X < Y < Z$). Клиент, зашедший в гостиницу, обладает некоторой суммой S денег и получает номер по своим финансовым возможностям, если тот свободен. Если среди доступных клиенту номеров нет свободных, клиент уходит искать ночлег в другое место, но на следующее утро возвращается. Если клиент получил номер, то он ночует и уходит. Создать многопоточное приложение, моделирующее работу гостиницы в течение месяца. Для отсчета каждого часа завести отдельный поток BigBen. Клиенты появляются в гостинице в случайное время и днем, и ночью, группами и поодиночке.

3.26. *Вторая задача о гостинице.* В гостинице N одноместных номеров и M двухместных. В гостиницу приходят одинокие клиенты-дамы и одинокие клиенты-джентльмены, которые могут провести ночь в одноместном номере или в номере только с представителем своего пола. Кроме того, клиентами гостиницы могут также быть пары, желающие провести ночь вместе. Если для клиента не находится подходящего номера, он уходит искать ночлег в другое место, но на следующее утро возвращается. Если клиент получил номер, то он ночует и уходит. Создать многопоточное приложение, моделирующее работу гостиницы в течение месяца. Для отсчета каждого часа завести отдельный поток BigBen. Клиенты появляются в гостинице в случайное время и днем, и ночью, группами, парами и поодиночке.

3.27. *Задача о клумбе.* На клумбе растет N цветов, за ними непрерывно следят два садовника и поливают увядшие цветы; при этом оба садовника очень боятся полить один и тот же цветок. Создать многопоточное приложение, моделирующее состояния клумбы и действия садовников. Для изменения состояния каждого цветка создать отдельный поток.

3.28. *Задача о садовниках-интровертах.* Имеется пустой участок земли (двумерный массив) и план сада, который необходимо реализовать. Эту задачу выполняют два садовника, которые не хотят встречаться друг с другом. Первый садовник начинает работу с верхнего левого угла сада и всегда перемещается слева направо. Закончив обработку очередного ряда, он спускается на один ряд вниз и начинает обрабатывать его. Второй садовник начинает работу с нижнего правого угла сада и всегда перемещается снизу вверх. Закончив обработку очередного ряда, он перемещается на один ряд влево и начинает обрабатывать его. Если садовник видит, что участок сада уже возделан другим садовником, он идет дальше. Если садовник встречает своего напарника, то он в испуге возвращается на два шага назад и отдыхает некоторое время, переживая увиденное. Садовники должны работать параллельно. Создать многопоточное приложение, моделирующее работу садовников.

3.29. *Задача о картинной галерее.* Вахтер следит за тем, чтобы в картинной галерее было не более N посетителей. Для обозрения представлены M картин. Посетитель ходит от картины к картине (в произвольном порядке), и если на картину любуются более чем десять посетителей, он стоит в стороне и ждет, пока число желающих увидеть картину не станет меньше. Посетитель может покинуть галерею, только осмотрев все картины. Создать многопоточное приложение, моделирующее работу картинной галереи. Посетители появляются в галерее в случайное время группами и поодиночке.

3.30. *Задача о болтунах.* N болтунов имеют телефоны, ждут звонков и звонят друг другу, чтобы побеседовать. Если телефон занят, болтун будет звонить по этому номеру, пока ему не ответят. Побеседовав, болтун не унимается: он некоторое время ждет звонка и начинает звонить на другой номер. Создать многопоточное приложение, моделирующее поведение болтунов. Предусмотреть механизм, предотвращающий мертвую блокировку, т. е. ситуацию, при которой все болтуны пытаются дозвониться друг другу и, соответственно, не могут ответить на звонки.

3.31. *Задача о программистах.* В отделе работают N программистов. Каждый программист пишет свою программу и отдает ее на проверку другому. Программист проверяет чужую программу, когда его собственная уже написана. По завершении проверки он дает ответ: программа написана правильно или неправильно. Программист спит, если не пишет свою программу и не проверяет чужую. Программист просыпается, когда получает заключение от другого программиста или программу на проверку. Если программа признана правильной, он начинает писать другую программу; если программа признана неправильной, он исправляет ее и отправляет на

проверку тому же программисту, который ее проверял ранее. Создать многопоточное приложение, моделирующее работу программистов.

3.32. Изготовление знаменитого самурайского меча – катаны – происходит в три этапа. Сначала младший ученик мастера выковывает заготовку будущего меча. Затем старший ученик мастера закаливает меч в трех водах – кипящей, студеной и теплой. И в конце мастер собственноручно изготавливает рукоять меча и наносит узоры. Мастер никогда не разрешает начинать изготовление нового меча, пока полностью не закончится изготовление предыдущего, а чтобы ученики не пребывали в праздности, пока они не изготавливают очередной меч, они должны считать камни в Саду камней. Перед началом изготовления каждого нового меча каждый ученик сообщает мастеру, сколько камней он насчитал. Требуется создать многопоточное приложение, моделирующее процесс создания 100 мечей, в приложении мастер и его ученики представлены разными потоками.

3.33–3.58. Сделать многопоточное приложение из задачи 3.7–3.32 многопроцессным. Пришлось ли вам изменять механизмы синхронизации? Почему?

Глава 4 | ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ OpenMP

В главе обсуждается механизм реализации многопоточности на основе технологии OpenMP – распределение нагрузки между потоками при обработке циклов, организация параллельной нециклической обработки и методы синхронизации потоков, а также некоторые дополнительные возможности библиотеки OpenMP. Рассматривается использование OpenMP в языках Си или Си++.

4.1. Программная модель OpenMP

Как уже обсуждалось в гл. 1, для введения многопоточности в последовательные языки программирования можно использовать механизмы, предоставляемые ОС, или специальные библиотеки. В предыдущей главе описаны средства организации многопоточности в ОС Windows. Аналогичный POSIX-интерфейс для организации потоков (например, библиотека многопоточного программирования Pthreads) поддерживается на всех UNIX-системах.

Подход к созданию многопоточных приложений с помощью WinAPI или Pthreads можно назвать низкоуровневым, поскольку создание и управление потоками, а также их синхронизация описываются в приложениях явно. Более высокоуровневые механизмы многопоточности предоставляются объектно ориентированными библиотеками (например, Qt, Intel® Threading Building Blocks). Одним из высокоуровневых подходов к созданию многопоточного приложения является также технология, основанная на директивах компилятора.

Многие поставщики SMP-архитектур (Sun, HP, SGI) в своих компиляторах поддерживают спецдирективы для распараллеливания циклов. Однако эти наборы директив, во-первых, весьма ограничены, а во-вторых, несовместимы между собой. Технология OpenMP является во многом обобщением и расширением упомянутых наборов директив.

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах в модели общей памяти. Первый стандарт OpenMP [80, 89] был разработан в 1997 году как API, ориентированный на написание переносимых многопоточных приложений. Сначала он был основан на языке Fortran, но уже в 1998 году вышли версии для языков Си и Си++. В настоящее время используется спецификация стандарта

OpenMP v. 3.1, принятая в июле 2011 года. Изложение материала в настоящей главе основано на этой версии стандарта (см. также монографии и учебные пособия [2, 12, 13, 54, 91]). В июле 2013 года был выпущен следующий набор спецификаций стандарта OpenMP v. 4.0. В конце главы кратко обсуждаются новые возможности этой спецификации.

OpenMP предоставляет простой способ создания потоков в приложениях, не требуя от программиста глубоких знаний о создании, синхронизации и уничтожении потоков, не всегда даже актуально знание о том, сколько потоков следует создать. Разработчикам программ OpenMP предоставляет независимый от платформы набор директив (Directives), функций библиотеки OpenMP периода выполнения (Runtime Library Routines) и переменных среды (Internal Control Variables, Environment Variables), которые в совокупности указывают компилятору, как и где именно следует вставить потоки в приложение.

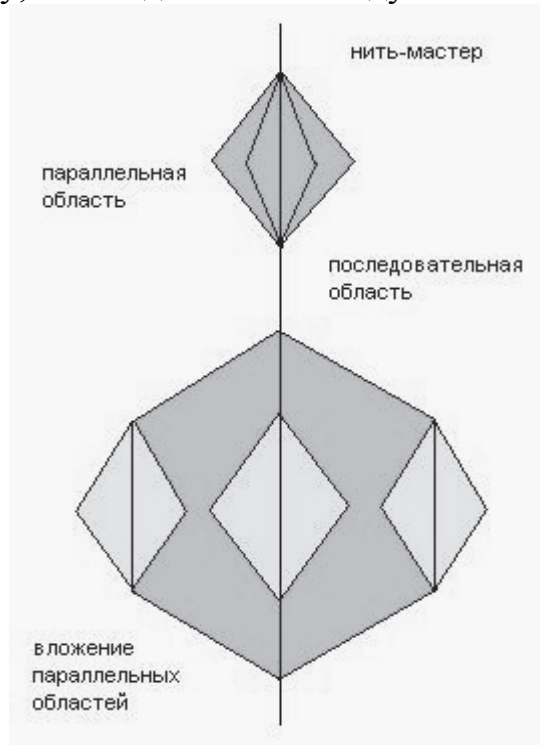


Рис. 4.1. OpenMP-программа

Простота использования такого подхода подкупает – большинство циклов можно распараллелить, вставив всего одну директиву непосредственно перед циклом. Однако следует отметить, что в настоящее время технология OpenMP является мощным, сложным и разветвленным средством программирования ВС с общей памятью. Возможности написания эффективного параллельного кода на OpenMP далеко не ограничиваются вставкой пары директив в последовательную версию.

Технология OpenMP представляет программу (рис. 4.1) как совокупность параллельных и последовательных областей. В момент запуска программы порождается *нить-мастер*¹ (основная нить), которая начинает выполнение со стартовой точки. В *последовательной*

области программы всегда выполняется только основная нить. Для поддержки параллелизма используется схема FORK/JOIN. При входе в *парал-*

¹ Согласно общепринятой терминологии в OpenMP потоки называются нитями. Далее в этой главе мы будем придерживаться этого термина.

тельную область нить-мастер порождает *группу* дополнительных нитей (FORK). Все порожденные нити, включая нить-мастер, исполняют один и тот же код параллельной области. Когда нити группы завершают инструкции в параллельной области, они синхронизируются барьером (JOIN) и закрываются, оставляя только нить-мастер. В параллельную область могут быть вложены другие параллельные области, в которых каждая нить первоначальной области становится мастером для соответствующей группы нитей.

Для возможности использования механизмов OpenMP необходимо скомпилировать программу компилятором, поддерживающим OpenMP, с указанием соответствующего ключа (табл. 4.1).

Таблица 4.1

Компилятор	Ключ компиляции
gcc	-fopenmp
Intel® C/C++ (icc): Linux, MacOSX Windows	-openmp /Qopenmp
Visual C++	/openmp

Для указания компилятору, как организовать параллельное выполнение некоторого блока программного кода, применяются специальные *директивы*. В языках Си/Си++ используется стандартная конструкция препроцессора `#pragma`, которая дополняется обязательным ключевым словом `omp`, названием директивы с указанием при необходимости одной или нескольких опций:

```
#pragma omp <директива> [опция [,] опция]...
```

Компилятор интерпретирует директивы OpenMP и создает параллельный код. При использовании компиляторов, не поддерживающих OpenMP, директивы OpenMP игнорируются без дополнительных сообщений, что обеспечивает корректную компиляцию программы в любом случае.

Функции OpenMP служат для получения информации о параметрах среды исполнения, а также для изменения этих параметров во время исполнения `omp`-программы. Для того чтобы использовать `omp`-функции, в программу необходимо включить заголовочный файл `omp.h`. Если компилятор не поддерживает OpenMP, то использование `omp`-функций приводит к ошибке компиляции. Для того чтобы программа, использующая функции OpenMP, могла оставаться корректной для обычного компилятора, можно прилинковать специальную библиотеку, которая определит для каждой функции соответствующую «заглушку» (`stub`). Например, в ком-

пиляторе Intel® C/C++ соответствующая библиотека подключается заданием ключа `-openmp-stubs`.

Компилятор с поддержкой OpenMP определяет макрос `_OPENMP`, который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы. Для условной компиляции вызовы `omp`-функций следует обрамлять директивой условной компиляции:

```
#ifdef _OPENMP
    вызов omp-функции
#endif
```

Пример 4.1 содержит код простейшей программы, проверяющей, поддерживает ли компилятор OpenMP.

Пример 4.1. Условная компиляция

```
#include <stdio.h>
int main() {
    #ifdef _OPENMP
        printf("OpenMP is supported!\n");
    #elif
        printf("OpenMP is't supported!\n");
    #endif
}
```

4.2. Модель памяти OpenMP

В распоряжение программиста OpenMP предоставляет целую иерархию памяти (рис. 4.2). Поскольку программа выполняется в среде с общей памятью, то все нити параллельной области имеют к ней доступ. Кроме того, данные могут храниться в локальной памяти нити. Такие данные в общем случае уничтожаются при выходе программы из параллельной области. Существует также механизм сохранения локальных данных нитей между параллельными областями.

Переменные декларируются и инициализируются вне параллельной области.

При входе в параллельную область часть переменных может оставаться общими (*shared*). Общая переменная всегда существует в одном экземпляре, к ней имеют асинхронный неатомарный доступ все нити. Если не предпринять никаких специальных действий по синхронизации доступа к общей переменной, то, как уже неоднократно обсуждалось, корректным и безопасным будет только одновременное чтение этой переменной

несколькими нитями; иначе возникает ситуация «гонки данных», при этом значение, которое будет храниться в каждый момент времени в общей переменной, не детерминировано.

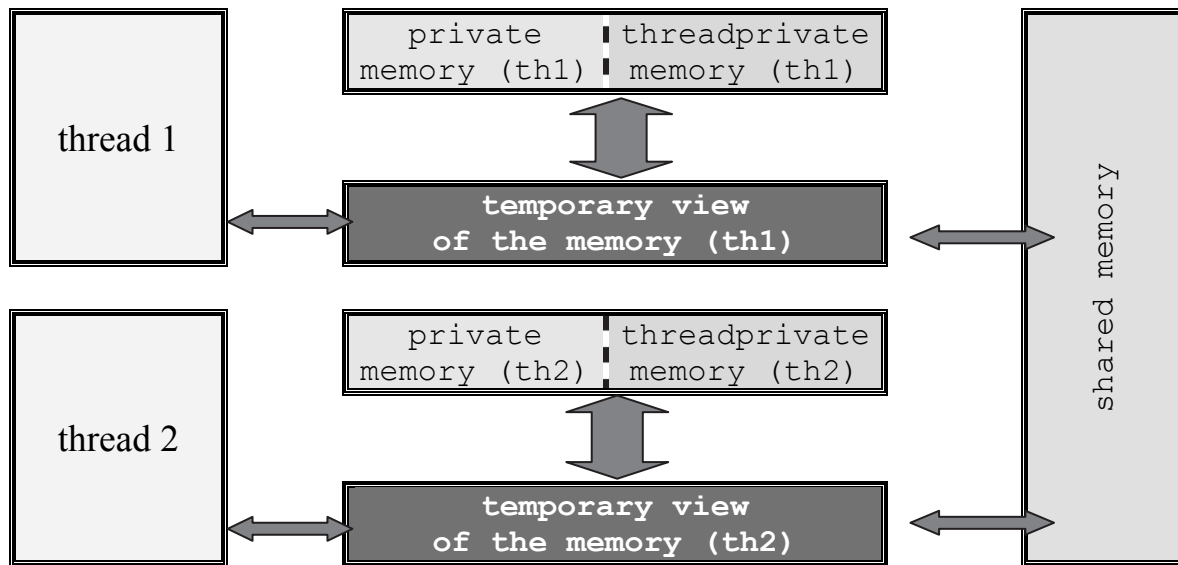


Рис. 4.2. Модель памяти OpenMP

Для другой части переменных при входе в параллельную область в каждой нити порождается локальная копия, к которой нить имеет монопольный доступ. Такие переменные называются локальными (*private*) для нити. Копии локальных переменных имеют такое же имя, тип и размер, что и соответствующая переменная в последовательной области. При входе в параллельную область локальные переменные инициализируются значениями согласно правилам по умолчанию или явно указанным в директиве, порождающей параллельную область. Изменение нитью значения своей локальной переменной никак не влияет на изменение значения этой же локальной переменной в других нитях.

В общем случае локальные копии переменных уничтожаются, когда программа завершает параллельную область. Значение, которое будет иметь соответствующая переменная в последовательной области, определяется реализацией или непосредственно в опциях директивы, породившей параллельную область.

Если количество нитей в нескольких параллельных областях одинаково и исключена вложенность параллельных областей, то можно объявить специальные статические переменные (*threadprivate*). Для них при входе в первую параллельную область в каждой нити создается локальная копия, которая не уничтожается в последовательной области и сохраняет локальные значения, доступные нити с тем же номером в следующей параллельной области.

Необходимо понимать, что каждая нить при работе с памятью имеет свой локальный буфер, кэш, регистры, где временно хранит данные. Это временное представление памяти нити (*temporary view of the memory*) не связано непосредственно с технологией OpenMP, а является аппаратной и/или программной надстройкой. Если переменная локальная, то временное представление памяти не вызывает конфликтов и может не учитываться программистом. Собственно при последовательном программировании вы уверены, что если по алгоритму необходимо сначала записать новое значение переменной, а потом использовать его в вычислениях, то к тому времени, как переменная понадобится по чтению, ее значение будет обновлено. Как уже неоднократно обсуждалось, при параллельном использовании одной переменной несколькими нитями необходимо учитывать правила согласования памяти.

В OpenMP принята ослабленная модель согласования (*relaxed-consistency shared-memory model*), т. е. возможна временная несогласованность представления памяти нити и оперативной памяти, а порядок согласования разными нитями своего представления и оперативной памяти зависит только от кода самой нити и не учитывает их взаимное влияние. Переменная буферизуется во временном представлении памяти нити и может быть записана в оперативную память позже.

В OpenMP предусмотрена специальная операция принудительной синхронизации временного представления памяти нити и оперативной памяти. Неявная синхронизация памяти происходит также при выполнении некоторых директив, например, барьерной синхронизации.

Более подробно модель согласования обсуждается в разделе, посвященном директиве `omp flush`.

4.3. Среда выполнения OpenMP-программы

Внутренние контрольные переменные OpenMP-программы

Для обеспечения переносимости параллельных программ и управления выполнением параллельных областей в OpenMP определен ряд *внутренних контрольных переменных* (*internal control variables*, ICV, далее IC-переменных), определяющих поведение приложения. Некоторые IC-переменные являются глобальными (*global*), т. е. существует только одна их копия на всю программу. Изменение значений глобальных IC-переменных возможно только вне параллельной области. Часть IC-переменных имеет по одной копии для каждой параллельной области (*data environment*). Значения таких переменных могут быть изменены в любой части программы, в том числе в параллельной области.

Существует несколько способов задания значений внутренних контрольных переменных.

Во-первых, значения большинства внутренних контрольных переменных в начале программы инициализируются соответствующими значениями *переменных среды выполнения* (environment variables), которые устанавливаются до начала выполнения программы. Например, в ОС Linux в командной оболочке `bash` это можно сделать при помощи команды `export`. Следующая команда устанавливает количество нитей, которые будут созданы по умолчанию при входе в параллельную область:

```
export OMP_NUM_THREADS=n
```

Во-вторых, значения некоторых IS-переменных можно установить во время выполнения программы с помощью функций времени выполнения. Имена таких функций начинаются на `omp_set_`. Значения IS-переменных, установленные с помощью функций, имеют приоритет над переменными среды выполнения, установленными по умолчанию. Если область видимости IS-переменной – вся программа, то функции `omp_set_` можно вызывать только вне параллельных областей. Если область видимости IS-переменной – параллельная область, то значение, установленное в функции, начинает действовать для следующей за вызовом функции параллельной области.

В-третьих, значения IS-переменных могут быть установлены непосредственно в опции директивы, создающей параллельную область. Такое определение переменных имеет максимальный приоритет, но его действие распространяется только на параллельную область действия директивы.

Функции, работающие с IS-переменными, и соответствующие переменные среды выполнения

В OpenMP предусмотрен большой набор библиотечных функций времени выполнения, которые выполняют следующие действия:

- контролируют и запрашивают различные параметры, определяющие поведение приложения (число нитей или процессоров, возможность вложенного параллелизма или динамического изменения количества исполняемых нитей и т. д.). Эти функции имеют префикс `omp_get_`;
- назначают значения IS-переменным, которые при выполнении приложения имеют приоритет над соответствующими переменными среды, заданными по умолчанию. Такие функции имеют префикс `omp_set_`;
- работают с системным таймером;
- реализуют механизм синхронизации нитей на основе замков (locks).

Перечислим основные функции.

```
int omp_in_parallel(void);
```

Функция возвращает значение `true`, если вызвавшая ее нить выполняет параллельную область, и `false` в противном случае.

```
int omp_get_num_procs(void);
```

Функция возвращает число процессоров, доступных программе.

```
int omp_get_num_threads(void);
```

Функция возвращает число нитей, входящих в текущую группу. Если вызывающая нить выполняется вне параллельной области, функция возвращает 1. Соответствующая переменная среды выполнения `OMP_NUM_THREADS`.

```
int omp_get_thread_num(void);
```

Функция возвращает номер вызвавшей ее нити в текущей группе. Функция возвращает значение в диапазоне от 0 до `(omp_get_num_threads() - 1)`.

```
int omp_get_thread_limit(void);
```

Функция возвращает максимально возможное количество нитей в любой параллельной области программы. Область действия – вся программа. Соответствующая переменная среды выполнения `OMP_THREAD_LIMIT`, значение которой, как правило, определяется аппаратными ограничениями.

```
int omp_get_max_threads(void);
```

Функция возвращает число нитей, которые будут созданы в ближайшей параллельной области, если это не будет переопределено параметрами директивы. По умолчанию функция возвращает число нитей, определенное ближайшей выполненной функцией `omp_set_num_threads()`, или значение переменной `OMP_NUM_THREADS`, или максимальное число нитей, поддерживаемых аппаратно (значения указаны в порядке уменьшения приоритета).

Число нитей, которое будет создано для выполнения ближайшей параллельной области, зависит от нескольких переменных среды выполнения (поддержки динамического создания потоков `OMP_DYNAMIC`, поддержки вложения областей `OMP_NESTED` и `OMP_NUM_THREADS`), а также значений IS-переменных, устанавливаемых перечисленными ниже функциями.

```
void omp_set_num_threads(int num_threads);
```

Функция устанавливает число нитей для выполнения следующей параллельной области, которая встретится с вызвавшей ее нитью, если это не будет переопределено параметрами директивы. Входной параметр `num_threads` – число нитей. Область действия – параллельная область. Ус-

установленное значение имеет приоритет над переменной среды выполнения `OMP_NUM_THREADS`.

```
void omp_set_dynamic (int dynamic_threads);
```

Функция устанавливает IS-переменную булевого типа, отвечающую за возможность динамического создания нитей. Если устанавливается значение `false`, то для выполнения следующей параллельной области исполняющая среда OpenMP создаст группу, число потоков в которой точно равно значению, возвращаемому функцией `omp_get_max_threads()`. Если устанавливается значение `true`, то исполняющая среда OpenMP создаст группу, которая может содержать переменное число потоков, не превышающее значение, возвращаемое функцией `omp_get_max_threads()`. В этом случае решение принимается на основе дополнительной информации о доступных программе вычислителях и определяется реализацией. Область действия функции – параллельная область. Установленное значение имеет приоритет над переменной среды выполнения `OMP_DYNAMIC`, которая по умолчанию равна `false`.

```
int omp_get_dynamic(void);
```

Функция возвращает текущее значение IS-переменной, отвечающей за возможность динамического создания потоков. Область действия – параллельная область. Соответствующая переменная среды выполнения `OMP_DYNAMIC`.

```
void omp_set_nested(int nested_threads);
```

Функция устанавливает IS-переменную булевого типа, отвечающую за возможность вложения параллельных областей. Если входной параметр `nested_threads` имеет значение `true`, то нить, уже выполняющая параллельную область и встречающая директиву создания другой параллельной области, создает новую группу нитей, в противном случае формируется группа из одной нити. Область действия – параллельная область. Установленное значение имеет приоритет над переменной среды выполнения `OMP_NESTED`, которая по умолчанию равна `false`.

```
int omp_get_nested(void);
```

Функция возвращает текущее значение параметра, отвечающего за возможность вложения параллельных областей. Область действия – параллельная область. Соответствующая переменная среды выполнения `OMP_NESTED`.

```
void omp_set_max_active_levels(int max_active_levels);
```

Функция устанавливает IS-переменную целого типа, определяющую максимально возможное вложение параллельных областей. Область действия – вся программа. Установленное значение имеет приоритет над переменной среды выполнения `OMP_MAX_ACTIVE_LEVELS`.

```
int omp_get_max_active_levels(void);
```

Функция возвращает максимально возможное количество уровней вложенности в любой параллельной области программы. Область действия – вся программа. Соответствующая переменная среды выполнения `OMP_MAX_ACTIVE_LEVELS`.

```
int omp_get_level(void);
```

Для вызвавшей ее нити функция возвращает текущий уровень вложенности параллельной области. Область действия – параллельная область.

```
int omp_get_active_level(void);
```

Для вызвавшей ее нити функция возвращает текущий уровень вложенности параллельной области, исключая параллельные области, состоящие из одной нити. Область действия – параллельная область.

```
int omp_get_ancestor_thread_num(int level);
```

Для вызвавшей ее нити функция возвращает номер нити, породившей текущую на уровне вложенности `level`. Значение `level` должно лежать в диапазоне `[0, omp_get_level]`; иначе функция возвращает значение -1. Если функция вызывается с `level=0`, то возвращается значение 0. Если `level = omp_get_level`, то результат эквивалентен вызову функции `omp_get_thread_num()`. Область действия – параллельная область.

```
int omp_get_team_size(int level);
```

Для вызвавшей ее нити функция возвращает количество нитей в группе, породившей текущую на уровне вложенности `level`. Значение `level` должно лежать в диапазоне `[0, omp_get_level]`, иначе функция возвращает значение -1. Если функция вызывается с `level=0`, то возвращается значение 1. Если `level = omp_get_level`, то результат эквивалентен вызову функции `omp_get_num_threads()`. Область действия – параллельная область.

Примеры 4.28–4.30 демонстрируют использование рассмотренных функций среды выполнения OpenMP.

Функции замера времени

Проблема правильного замера времени выполнения многопоточной программы является весьма актуальной. При работе со специальными библиотеками, например OpenMP, всегда удобнее использовать средства, которые предоставляются самой библиотекой. В OpenMP предусмотрены следующие функции для работы с системным таймером (пример 4.2).

```
double omp_get_wtime(void);
```

Функция возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом.

Если некоторый участок кода окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, наступил до запуска процесса и не будет изменён за время его существования. Таймеры разных нитей могут быть не синхронизированы и выдавать различные значения.

```
double omp_get_wtick(void);
```

Функция возвращает в вызвавшей нити разрешение таймера в секундах. Это время можно рассматривать как меру точности таймера.

В примере 4.2 разность времён двух следующих друг за другом замеров времени даёт время, затрачиваемое процессом на замер времени. Также выводится точность системного таймера.

Пример 4.2. Замер времени с помощью функций OpenMP

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv) {
    double start_time, end_time, tick;
    start_time = omp_get_wtime();
    end_time = omp_get_wtime();
    tick = omp_get_wtick();
    printf("Время на замер времени %lf\n", end_time-
start_time);
    printf("Точность таймера %lf\n", tick);
    return 0;
}
```

Часть функций времени выполнения, IS-переменных и переменных среды будут описаны далее, в разделах, связанных с распределением работы между нитями внутри параллельной области и с синхронизацией нитей.

4.4. Директива `omp parallel`

Создание параллельной области

Директива `omp parallel` создает параллельную область для следующего за ней структурного блока. Общий синтаксис директивы имеет следующий вид:

```
#pragma omp parallel [опция[,] опция]...  
    структурный блок
```

Директива сообщает компилятору, что структурный блок кода должен быть выполнен параллельно, несколькими нитями. Количество порождаемых нитей определяется соответствующей IS-переменной. Начальное значение этой переменной задается перед запуском программы, хранится в переменной среды окружения `OMP_NUM_THREADS` и может изменяться в ходе ее выполнения с помощью функции `omp_set_num_threads()`. Узнать это значение до параллельной области можно с помощью функции `omp_get_max_threads()`. (Порождается `(omp_get_max_threads() - 1)` нить, поскольку нить-мастер уже существует.)

Следует понимать, что каждая нить асинхронно будет выполнять один и тот же поток команд, но, возможно, не один и тот же набор команд — все зависит от операторов, управляющих логикой программы.

Отдавая дань традиции, приведем в качестве первого примера программы, использующей технологию OpenMP, аналог программы `Hello World` для двух нитей (пример 4.3).

Пример 4.3. Простейшая программа с использованием OpenMP

```
#include <stdio.h>  
int main(){  
    #pragma omp parallel num_threads(2)  
    {  
        printf("Hello World!\n");  
    }  
}
```

В примере каждая нить должна вывести приветствие один раз; соответственно подразумевается, что вывод программы в окно консоли должен выглядеть следующим образом:

```
Hello World!  
Hello World!
```

Однако поскольку нити выполняются асинхронно и параллельно и, как мы уже знаем, функция вывода на консоль в Си не атомарна, то возможно возникновение ситуации «гонок» и результат может оказаться, например, таким:


```
HellHell oo WorWlodrl
d!!
```

Следует отметить, что такие ошибки кода являются недетерминированными, а потому найти их крайне трудно, а OpenMP не предоставляет механизмов автоматического поиска и исключения подобных ситуаций.

Опции директивы `omp parallel`

В директиве `omp parallel` могут быть использованы следующие опции:

- `if (<выражение>)` – если выражение истинно, код блока выполняется параллельно, иначе код выполняется последовательно;
- `num_threads (<целочисленное выражение>)` – явное задание количества нитей, которые будут выполнять параллельную область. По умолчанию выбирается последнее значение, установленное с помощью функции `omp_set_num_threads()`, или значение переменной среды выполнения `OMP_NUM_THREADS`;
- `shared (<list_item>)` – переменные, перечисленные в списке опции `shared`, разделяются всеми потоками программы;
- `private <list_item>` – для переменных, перечисленных в списке опции `private`, в каждой нити порождается локальная копия. Начальные значения этих переменных не определены. В конце параллельной области такие переменные уничтожаются;
- `firstprivate (<list_item>)` – для переменных, перечисленных в списке опции `firstprivate`, в каждой нити порождается локальная копия, инициализирующаяся значением соответствующей переменной нити-мастера;
- `default (shared | none)` – опция определяет, являются ли переменные общими по умолчанию (`shared`) или тип их использования необходимо в обязательном порядке определить (`none`);
- `copyin (<list_item>)` – переменные, перечисленные в списке опции `copyin`, должны быть объявлены как `threadprivate`. При входе в параллельную область значения этих переменных инициализируются соответствующим значением в нити-мастере. Особенности работы с `threadprivate`-переменными обсуждаются в примерах 4.5 и 4.6;
- `reduction (операция: <list_item>)` – в этой опции описываются переменная и операция, которую следует выполнить при завершении параллельной области над всеми локальными копиями этой переменной. Переменная должна быть скалярного типа. Поддерживаемые опцией `reduction` операции, а также инициализированные значения каждой копии переменной описаны в табл. 4.2. В конце параллельного блока кода опера-

тор опции `reduction` применяется к каждой частной копии переменной, а также к ее исходному значению. Пример 4.14 демонстрирует применение этой опции при распараллеливании цикла.

Таблица 4.2

Операции опции <code>reduction</code>	Инициализированное значение каждой копии <code>list_item</code>	Правило получение финального значения переменной <code>list_item</code>
<code>+</code>	0	<code>list_item += private_copy</code>
<code>*</code>	1	<code>list_item *= private_copy</code>
<code>-</code>	0	<code>list_item -= private_copy</code>
<code>&</code>	<code>~0</code> (каждый бит установлен)	<code>list_item &= private_copy</code>
<code> </code>	0	<code>list_item = private_copy</code>
<code>^</code>	0	<code>list_item ^= private_copy</code>
<code>&&</code>	1	<code>list_item &&= private_copy</code>
<code> </code>	0	<code>list_item = private_copy</code>
<code>max</code>	Наименьшее возможное значение для типа переменной <code>list_item</code>	<code>list_item = list_item > private_copy ? private_copy : list_item;</code>
<code>min</code>	Наибольшее возможное значение для типа переменной <code>list_item</code>	<code>list_item = list_item < private_copy ? private_copy : list_item;</code>

По умолчанию все переменные, порожденные вне параллельной области, в параллельной области являются общими (`shared`), т. е. доступны всем нитям. Еще раз подчеркнем, что если происходит асинхронная запись несколькими потоками в такую переменную или хотя бы один поток записывает в переменную данные и хотя бы один поток читает из нее, то результат этих операций недетерминирован (непредсказуем). Общими (`shared`) являются статические переменные (`static`), даже если они определены в параллельной области. Динамически выделенная память также является общей, в то же время указатель на динамическую переменную может быть локальным (`private`).

Локальными (`private`) по умолчанию всегда являются:

- индексы параллельных циклов `for`;
- переменные, объявленные в параллельных областях, за исключением статических переменных;
- любые переменные, указанные в разделах `private`, `firstprivate`, и `reduction`.

Пример 4.4 демонстрирует использование директивы `omp parallel` для прямой организации параллельных вычислений. В примере реализовано вычисление произведения двух квадратных матриц по полосам, описанное в примере 1.1 (гл. 1). В функции `main()` примера 4.4 динамически выделяется память для хранения исходных матриц `a` и `b`, а также матрицы произведения `c`. Здесь же матрицы инициализируются начальными значениями; с помощью функций среды выполнения запрещается динамически изменять количество потоков; явно устанавливается количество потоков в параллельной области, затем вызывается функция `mult()`.

Заметим, что в строке, помеченной в комментарии `(*)`, определено, что если заданное константой `n` количество строк в матрице меньше, чем заданное в `#define` количество нитей, то количество нитей переопределяется единицей и все формулы в программе остаются корректными, а параллельная область состоит из одной нити.

Пример 4.4. Параллельная область. Вычисление произведения матриц по полосам.

Ручная статическая балансировка нагрузки

```
#include <omp.h>
#include <stdio.h>
#define NUM_OF_TH 8 //количество нитей
const int n = 1000; //размерность 2D-массива
//в одном направлении
//Функция вывода на консоль 2D-массива размерности nn x mm
void prn_2d (int* array, int nn, int mm){
    int ii, jj;
    for (ii=0;ii<nn;ii++){
        for (jj=0;jj<mm;jj++){
            printf("%d ",array[ii*n+jj]);printf("\n");
        }
        printf("\n");
    }
    //Функция умножения полосы строк массива a
    //на полосу столбцов массива b
    void mult_strip(int *xx, int *yy, int *zz,
                    int i_start, //индекс начала полосы
                    int i_strip_width){ //ширина полосы

        int ii,jj,kk;
        int i_finish=i_start+i_strip_width;
        for (ii=i_start; ii<i_finish; ii++){
            for (jj=0 ;jj<n;jj++){
                zz[ii*n+jj] = 0;
                for (kk=0 ;kk<n;kk++){
                    zz[ii*n+jj]+= xx[ii*n+kk] * yy[kk*n+jj];
                }
            }
        }
    }
}
```

Пример 4.4 (окончание). Параллельная область. Вычисление произведения матриц по полосам. Ручная статическая балансировка нагрузки

```
//Функция определения параллельной области
//и распределения работы по нитям
//для вычисления произведения матриц
void mult(int *x, int* y, int* z, int num_pts){
    int current_th, num_th, c1, c2;
    int current_start, current_strip_width;
    #pragma omp parallel default(shared) \
        private(current_th,current_start,current_strip_width)
    {
        num_th = omp_get_num_threads();
        c1 = num_pts / num_th;
        c2 = num_pts % num_th;
        printf("Thread %d is informing: num_th=%d, c1=%d,
                c2=%d\n\n",omp_get_thread_num(),num_th,c1,c2);
        current_th = omp_get_thread_num();
        if (current_th < c2) current_start = current_th*(c1+1);
        else current_start = c2 + current_th * c1;
        current_strip_width = c1 + (current_th < c2);
        printf("%d: start = %d    width = %d\n\n", current_th,
                current_start, current_strip_width);
        mult_strip(x,y,z,current_start, current_strip_width);
    }
}

int main() {
    int i,j;
    int *a,*b,*c,num_of_th=(n>=NUM_OF_TH?NUM_OF_TH:1); // (*)
    a=(int*)malloc(sizeof(int)*n*n);
    if(!a){printf ("error in a-malloc\n");exit(1);}
    b=(int*)malloc(sizeof(int)*n*n);
    if(!b){printf ("error in b-malloc\n");exit(1);}
    c=(int*)malloc(sizeof(int)*n*n);
    if(!c){printf ("error in amin-malloc\n");exit(1);}
    for (i=0;i<n;i++)//заполнение массивов a и b
        for (j=0;j<n;j++){
            a[i*n+j] = (i*j)%7;
            b[i*n+j] = 0; if (i==j) b[i*n+j]=1;
        }
    printf("A\n"); prn_2d(a,n,n);
    printf("B\n"); prn_2d(b,n,n);
    printf("C\n"); prn_2d(c,n,n);
    omp_set_dynamic(0);
    omp_set_num_threads(num_of_th);
    mult(a,b,c,n);
    printf("C\n"); prn_2d(c,n,n);
    free(a); free(b); free(c)
    return 0;
}
```

В функции `mult()` для организации вычисления произведения матриц по полосам с помощью директивы `omp parallel` создается параллельная область, и на основе информации о размерности матрицы и количестве нитей в параллельной области для каждого номера нити вычисляются начальный номер строки в полосе и количество строк в полосе. Затем в параллельной области вызывается функция `mult_strip()` для непосредственного умножения полосы матрицы `a` на всю матрицу `b`.

Заметим, что директива `OpenMP` вместе с необходимыми опциями располагается в отдельной строке кода. Эта строка не должна содержать больше никаких операторов и выражений языка Си, включая операторные скобки `{ и }`. Если директива располагается на нескольких строках, то каждая строка, кроме последней, должна заканчиваться знаком `\`.

Функция `prn_2d()` выводит двумерный массив на консоль. В нашем примере она всегда вызывается в последовательной области и не имеет побочных эффектов.

Поскольку в параллельном программировании часто необходимо делить вычислительную нагрузку по нескольким вычислителям, в примере демонстрируется балансировка нагрузки в случае, когда количество данных (в нашем случае `num_pts` строк матрицы `a`) не кратно количеству доступных вычислителей (`num_th` нитей). Логично разделить строки между нитями как можно более равномерно, т. е. некоторые нити обработают максимум на одну строку больше, чем остальные. Пусть `int c1 = num_pts / num_th;` и `int c2 = num_pts % num_th.` Тогда `(num_th - c2)` нитей получат по `c1` строк, а `c2` нитей – по `c1+1` строк.

Механизм `threadprivate`-переменных

В языке Си статические переменные не видны за пределами файла или функции, в которых они объявлены, но сохраняют свои значения между вызовами. В `OpenMP` предусмотрен аналогичный механизм, который позволяет сохранять значения локальных копий переменных между параллельными областями. Для этого необходимо объявить соответствующие переменные как статические или переменные области видимости файла и включить их в список переменных специальной директивы `threadprivate`:

```
#pragma omp threadprivate (список переменных)
```

Заметим, что директива `threadprivate` является примером одной из небольшого количества *декларативных* директив `OpenMP`. Декларативные директивы не связаны с каким-либо блоком кода программы, однако связаны с пользовательскими объявлениями и должны располагаться только там, где язык программирования допускает декларации.

Пример 4.5 демонстрирует использование механизма `threadprivate`. Глобальные переменные `n1` и `n2` объявлены как `threadprivate`-переменные.

Пример 4.5. Использование механизмов `threadprivate` и `copyin`.

```
#include <omp.h>
#include <stdio.h>
int n1,n2;
#pragma omp threadprivate(n1,n2)
int main(int argc, char **argv){
    int num;
    n1=n2=1;
    #pragma omp parallel private (num)
    {
        num=omp_get_thread_num();
        printf("Out1, thread %d: value n1=%d; value n2=%d \n",
            num, n1,n2);
        n1=omp_get_thread_num();
        n2=omp_get_thread_num()+10;
        printf("Out2, thread %d: value n1=%d; value n2=%d \n",
            num, n1,n2);
    }
    printf("Out3, Master: value n1=%d; value n2=%d \n",
        n1,n2);
    #pragma omp parallel private (num) copyin(n2)
    {
        num=omp_get_thread_num();
        printf("Out4, thread %d: value n1=%d; value n2=%d \n",
            num, n1,n2);
    }
    return 0;
}
```

Значения этих переменных выводятся в четырёх разных местах программы. В начале функции `main()` обе переменные инициализируются единицами. При входе в первую параллельную область не предпринимается никаких действий по инициализации их локальных копий, поэтому в первом выводе «Out1...» нить-мастер выведет значения этих переменных равными 1, а все другие нити в зависимости от реализации равными либо нулю, либо произвольному значению. Второй вывод «Out2...» происходит после присвоения нитями значений локальным копиям `threadprivate`-переменным. Каждая нить в `n1` хранит свой номер, а `n2 = n1+10`. Затем в последовательной области будут ещё раз выведены «Out3...» — значения переменных `n1=0` (равные порядковому номеру нити-мастера) и `n2=10`. При входе в новую параллельную секцию по умол-

чанию каждая нить получает сохранившуюся между параллельными секциями копию переменной `n1` со значением, равным номеру нити, а значения копий переменных `n2` с помощью опции `copyin` директивы `parallel` инициализируются значением этой переменной в последовательной области, т. е. значением 10-й нити-мастера. Это подтверждает четвертый вывод «Out4...» переменных.

Пример 4.6 содержит код программы вычисления произведения двух квадратных матриц по полосам. Этот пример использует такую же схему распределения строк матрицы `a` по нитям, как и пример 4.4. Однако расчет параметров распределения выделен в отдельную функцию `work_distribution()`, которая вызывается из последовательной области программы. Функция вычисляет общие параметры `c1` и `c2`, исходя из размерности матрицы `num_pts` и количества нитей `num_th`. Затем в параллельной области каждая нить, исходя из своего номера `current_th`, рассчитывает параметры своей полосы (номер строки начала полосы `current_start` и ширину полосы `current_strip_width`), которые и сохраняются в `threadprivate`-памяти нити. Код функции `mult_strip()` остается без изменений (см. пример 4.4), однако функция вызывается из параллельной области `main()`. Код функции `prn_2d()` совпадает с соответствующим кодом примера 4.4.

4.5. Распределение работы в параллельной области по нитям

В параллельной области в общем случае все нити выполняют один и тот же код. Однако существует несколько вариантов распределения работы между запущенными нитями.

В примерах 4.4 и 4.6 использовался нехарактерный для OpenMP прием явного распределения работы между нитями. Вместо такого низкоуровневого подхода в распоряжение программиста OpenMP предоставляет ряд высокоуровневых директив, позволяющих распределить работу в параллельной области более элегантно и эффективно.

Пример 4.6. Использование `threadprivate`. Вычисление произведения матриц по полосам

```
#include <omp.h>
#include <stdio.h>
#define NUM_OF_TH 8 //количество нитей
const int n = 1000; //размерность квадратной матрицы
                      //в одном направлении
int current_start=0, current_strip_width=0;
#pragma omp threadprivate (current_start, \
                           current_strip_width)
```


Пример 4.6 (окончание). Использование `threadprivate`. Вычисление произведения матриц по полосам

```
//Функция печати из Примера 4.4
void prn_2d (int* array, int nn, int mm);
//Функция умножения полосы строк матрицы a
//на полосу столбцов матрицы b из Примера 4.4
void mult_strip(int *xx, int *yy, int *zz, int i_start,
                int i_strip_width);
//Функция распределения работы по нитям
void work_distribution(int num_pts, int num_th){
    int current_th, c1, c2;
    c1 = num_pts / num_th;
    c2 = num_pts % num_th;
    printf("%d: num_th = %d: c1 = %d    c2 = %d\n\n",
           omp_get_thread_num(), num_th, c1, c2);
    #pragma omp parallel default(shared) private(current_th)
    {
        current_th = omp_get_thread_num();
        if (current_th < c2) current_start = current_th * (c1 + 1);
        else current_start = c2 + current_th * c1;
        current_strip_width = c1 + (current_th < c2);
        printf("%d: start = %d width = %d\n\n",
               current_th, current_start, current_strip_width);
    }
}
int main(){
    int i, j;
    int *a, *b, *c,
    int num_of_th = (n >= NUM_OF_TH ? NUM_OF_TH : 1);
    a = (int*) malloc(sizeof(int) * n * n);
    if(!a){printf ("error in a-malloc\n"); exit(1);}
    b = (int*) malloc(sizeof(int) * n * n);
    if(!b){printf ("error in b-malloc\n"); exit(1);}
    c = (int*) malloc(sizeof(int) * n * n);
    if(!c){printf ("error in amin-malloc\n"); exit(1);}
    for (i=0; i<n; i++){
        for (j=0; j<n; j++){
            a[i*n+j] = (i*j)%7;
            b[i*n+j] = 0; if (i==j) b[i*n+j]=1;
        }
    }
    printf("A\n"); prn_2d(a, n, n);
    printf("B\n"); prn_2d(b, n, n);
    printf("C\n"); prn_2d(c, n, n);
    omp_set_dynamic(0);
    omp_set_num_threads(num_of_th);
    work_distribution(n, num_th);
    #pragma omp parallel default(shared)
    {
        mult_strip(a, b, c, current_start, current_strip_width);
        printf("C\n"); prn_2d(c, n, n);
        free(a); free(b); free(c)
    }
    return 0;
}
```

Следует помнить, что конструкции распределения работ в OpenMP не порождают новых нитей и, следовательно, должны встречаться в параллельной области, т. е. в области действия директивы `omp parallel`.

Директива `omp section`

Конструкция `sections` используется для задания неитеративного параллелизма.

```
#pragma omp sections [опция[,] опция]...
{
  #pragma omp section
  { <структурный блок 1> }
  ...
  #pragma omp section
  { <структурный блок n> }
}
```

Каждый блок `omp section` директивы `omp sections` выполняется асинхронно одной нитью. Если нитей больше, чем структурных блоков, то выбор нитей, которые будут задействованы в вычислениях, определяется реализацией. Если нитей меньше, чем блоков, то каждая нить группы сначала получит на выполнение по одному блоку, затем по мере освобождения нити будут назначаться на выполнение оставшихся блоков. При этом заранее неизвестно и определяется реализацией, какие блоки и в каком порядке будут выполняться. Гарантируется, что если в ходе выполнения некоторого структурного блока работа нити была прервана, то продолжение выполнения этого структурного блока будет поручено нити с тем же номером. В конце конструкции по умолчанию предполагается неявная барьерная синхронизация.

Для директивы `omp sections` определены четыре опции, совпадающие с опциями директивы `omp parallel`:

- `private(<список переменных>);`
- `firstprivate(<список переменных>);`
- `lastprivate(<список переменных>);`
- `reduction(оператор: <список переменных>).`

Кроме того, для `omp sections` определена опция `nowait`, которая позволяет нитям, закончившим выполнение своего структурного блока, продолжать выполнение программы без синхронизации с остальными нитями. Заметим, что даже при использовании опции `nowait` неявная синхронизация в конце параллельной области все равно обязательно происходит.

OpenMP поддерживает сокращенный способ записи комбинации директив `omp parallel` и `omp sections` в виде директивы

```
#pragma omp parallel sections [опция[,] опция]...
{
    #pragma omp section
    { <структурный блок 1> }
    ...
    #pragma omp section
    { <структурный блок n> }
}
```

для которой допустимы все опции директивы `#pragma omp parallel`.

Директива `omp master`

Директива

```
#pragma omp master
    структурный блок, выполняемый только нитью-мастером
```

предписывает выполнение следующего за ней структурного блока только нитью-мастером. Остальные нити параллельной области просто пропускают данный участок кода и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает. Директива не предполагает никаких опций.

Напомним, что в примере 4.4 приведен код вычисления произведения двух квадратных матриц. В частности, функция `mult()` в параллельной области вычисляет параметры декомпозиции матрицы на полосы, и каждая нить вызывает функцию `mult_strip()` для вычисления произведения своей полосы матрицы `a` на всю матрицу `b`. В примере 4.7 директива `omp master` добавлена в функцию `mult()` примера 4.4. Здесь директива `omp master` используется для вывода на консоль сообщения о параметрах `c1` и `c2` декомпозиции. Несмотря на то, что вывод расположен в параллельной секции, сделает его только один раз именно нить-мастер, причем остальные нити не станут дожидаться этого события и будут продолжать выполнение с оператора, помеченного (*).

Директива `omp single`

В параллельных областях может встретиться блок кода, выполнить который должна только одна нить, причем не важно, с каким номером.

Для ограничения таких участков кода в OpenMP служит директива `#pragma omp single [опция[,] опция]...`
структурный блок, выполняемый только одной нитью

Структурный блок выполняется одной нитью, при этом по умолчанию все остальные нити ждут завершения выполнения этого участка кода, т. е. в отличие от директивы `omp master` в конце `omp single` предполагается неявная барьерная синхронизация.

Пример 4.7. Пример использования директивы `master` в функции `mult` из примера 4.4

```
void mult(int *x, int* y, int* z, int num_pts){
    int current_th, num_th, c1, c2,
    int current_start, current_strip_width;
    #pragma omp parallel default(shared) \
        private(current_th,current_start,current_strip_width)
    {
        num_th = omp_get_num_threads();
        c1 = num_pts / num_th;
        c2 = num_pts % num_th;
        #pragma omp master
        {
            printf("master is informing (id=%d): num_th=%d,
                c1=%d, c2=%d\n\n",
                omp_get_thread_num(), num_th, c1, c2);
        }
        current_th = omp_get_thread_num(); // (*)
        if (current_th < c2) current_start = current_th * (c1 + 1);
        else current_start = c2 + current_th * c1;
        current_strip_width = c1 + (current_th < c2);
        printf("%d: start = %d width = %d\n\n",
            current_th, current_start, current_strip_width);
        mult_strip(x, y, z, current_start, current_strip_width);
    }
}
```

Для директивы `omp single` определены следующие опции:

- `private(<список переменных>)` – определяет список локальных переменных структурного блока директивы;
- `firstprivate(<список переменных>)` – определяет список переменных, локальные копии которых будут инициализированы значениями нити, выполняющей структурный блок директивы `omp single`;
- `nowait` – отменяет предполагаемое по умолчанию ожидание всеми нитями завершения выполнения структурного блока директивы;
- `copyprivate(<список переменных>)` – после выполнения структурного блока директивы `omp single` с этой опцией новые значения переменных, указанных в списке, будут доступны всем одноименным локальным переменным (`private` и `firstprivate`), описанным в начале параллельной области. Опция не может использоваться совместно с опцией `nowait`.

Удобно применять опцию `copyprivate` для широковещательной рассылки значений, вычисленных в структурном блоке директивы `omp single`, всем остальным нитям параллельной области (пример 4.8).

Пример 4.8. Пример использования директивы `single` для широковещательной рассылки в функции `mult` примера 4.4

```
void mult(int *x, int* y, int* z, int num_pts){
    int current_th, num_th, c1=0, c2=0,
    int current_start, current_strip_width;
    #pragma omp parallel default(shared) \
        private(c1,c2,current_th, \
                current_start,current_strip_width)
    {
        num_th = omp_get_num_threads();
        #pragma omp single copyprivate(c1,c2)
        {
            c1 = num_pts / num_th;
            c2 = num_pts % num_th;
            printf("Thread %d: num_th=%d, c1=%d, c2=%d\n\n",
                omp_get_thread_num(), num_th, c1, c2);
        }
        current_th = omp_get_thread_num();
        if (current_th < c2) current_start = current_th * (c1 + 1);
        else current_start = c2 + current_th * c1;
        current_strip_width = c1 + (current_th < c2);
        printf("%d: start = %d    width = %d\n\n",
            current_th, current_start, current_strip_width);
        mult_strip(x, y, z, current_start, current_strip_width);
    }
}
```

В примере 4.8 вновь немного изменена функция `mult()` примера 4.4. Значения параметров `c1` и `c2` декомпозиции матрицы на полосы вычисляются в директиве `omp single` только одной нитью и передаются всем нитям, выполняющим параллельную область, поскольку переменные описаны в списке опции `copyprivate`.

Директива `omp for`

Технология OpenMP считается самой простой, удобной и естественной для распараллеливания циклов.

Для логического распределения итераций цикла по группе нитей в параллельной области используется специальная директива `omp for`, которая имеет следующий синтаксис:

```
#pragma omp for [опция[,] опция]...
цикл for
```

В директиве `omp for` могут быть определены следующие опции:

- `private`, `firstprivate`, `reduction` — эти опции определяются точно так же, как и в директиве `parallel`;

- `lastprivate (<список переменных>)` – переменные, перечисленные в списке опции `lastprivate`, являются локальными для каждой нити. После завершения параллельной области переменным списка `lastprivate` присваиваются значения из нити, выполнившей самую последнюю итерацию цикла. Опцию не следует использовать совместно с опцией `nowait`;

- `nowait` – отменяет принятое по умолчанию выполнение барьерной синхронизации в конце цикла. В общем случае использование опции `nowait` повышает эффективность распараллеливания, однако из-за возможности возникновения трудновывяемых ситуаций «гонок» и «тупиков» пользоваться этой опцией следует крайне осторожно;

- `ordered` – опция сообщает, что в цикле могут встречаться одноименные директивы, определяющие блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации цикла выполняются при последовательном исполнении;

- `schedule(<type>[, chunk])` – опция определяет способ распределения итераций цикла между группой нитей;

- `collapse(n)` – опция указывает, что `n` последовательных тесно вложенных циклов ассоциируются с данной директивой. Для циклов образуются общее пространство итераций, которое делится между нитями (пример 4.12). Если опция `collapse` не задана, то вся директива `omp for` относится только к одному непосредственно следующему за ней циклу.

Важной для эффективности распараллеливания циклов является опция `schedule`, задающая способ распределения итераций между нитями параллельной области. Эта опция имеет два параметра – тип распределения `type` и размер блока `chunk`. Параметр `type` может принимать следующие значения:

- `static` – итерации статически делятся на блоки указанного в опции размера и распределяются между нитями. По умолчанию `chunk=1`. Такое распределение эффективно, если время выполнения каждой итерации примерно одинаково;

- `dynamic` – итерации делятся на блоки указанного в опции размера (по умолчанию `chunk=1`), освободившаяся нить берет на исполнение первый из еще необработанных блоков, до тех пор пока не исчерпаются все итерации (парадигма «портфель задач»). Такое распределение эффективно, если время выполнения итераций сильно варьируется;

- `guided` – аналогичное `dynamic` распределение итераций. Размер блока `chunk` уменьшается с некоторого начального значения до указанного в опции размера блока (по умолчанию `chunk=1`), причем каждый раз для нити выделяется порция итераций, вычисленная по следующему правилу: `chunk = max(кол-во_нераспределенных_итераций/omp_get_num_threads(),`

число итераций). Размер первоначально выделяемого блока зависит от реализации;

- `auto` – режим распределения итераций выбирается компилятором и/или системой выполнения. Размер блока в этом случае не задается;

- `runtime` – режим распределения итераций выбирается на основании соответствующей IS-переменной, которая в начале программы инициализируется значением переменной среды выполнения `OMP_SCHEDULE`. Эти переменные имеют формат `type[,chunk]`, где `type` принимает одно из следующих значений: `dynamic`, `guided`, `runtime`, `auto`, а `chunk` – размер блока.

Для управления способом распределения итераций цикла по нитям в OpenMP определены две функции.

```
void omp_set_schedule(omp_sched_t type, int chunk);
```

Функция устанавливает способ распределения итераций цикла по нитям (`type`) и размер блока (`chunk`). Установленное значение имеет приоритет над переменной среды выполнения `OMP_SCHEDULE`.

```
void omp_get_schedule(omp_sched_t * type, int *chunk );
```

Функция возвращает тип распределения итераций цикла по нитям `type` и размер блока `chunk`, которые будут применены в следующей встретившейся директиве `omp for`. Соответствующая переменная среды выполнения `OMP_SCHEDULE`.

Поскольку циклы являются самыми распространенными конструкциями, где выполнение кода можно эффективно распараллелить, OpenMP поддерживает сокращенный способ записи комбинации директив `omp parallel` и `omp for` в виде директивы

```
#pragma omp parallel for [опция[, опция]...]
    цикл for
```

для которой допустимы все опции обеих директив, за исключением `nowait`.

Существуют ограничения на формат цикла `for`, для которого применяется директива `omp for`:

```
for ([целочисленный тип] i = инвариант цикла;
    i {<,>,<=,>=} инвариант цикла;
    i {+,-}= инвариант цикла)
```

Под инвариантом цикла понимается выражение, значение которого не меняется в ходе выполнения цикла. Переменная цикла должна иметь тип `signed integer`; беззнаковые целые числа, такие как `DWORD`, исполь-

зовать в качестве инварианта цикла нельзя. Размер блока итераций не должен меняться в ходе выполнения цикла. Кроме того, инкрементная часть цикла `for` должна быть либо целочисленным сложением, либо целочисленным вычитанием из инварианта цикла. При этом если в качестве операции сравнения используется $<$ или \leq , то инвариант должен увеличиваться при каждой итерации, а при использовании операции $>$ или \geq инвариант должен уменьшаться.

Ясно также, что корректная работа программы не должна зависеть от того, какая именно нить какую итерацию цикла выполняет. Нельзя использовать побочный выход из параллельного цикла, т. е. не разрешены переходы из цикла, за исключением оператора `exit`, который завершает работу всего приложения. Если используются операторы `goto` или `break`, они должны приводить к переходам только внутри цикла. Исключения должны перехватываться также внутри цикла.

Все ограничения введены для того, чтобы OpenMP мог при входе в цикл определить число итераций.

Цикл из примера 4.9 невозможно распараллелить по двум причинам. Во-первых, заранее невозможно узнать, сколько раз цикл будет выполнен (условие `dx < 0.001` не является инвариантом цикла). Во-вторых, цикл содержит *зависимость по управлению* между итерациями цикла (значение `dx` в каждой итерации цикла зависит от значения переменной `x2`, вычисленной на предыдущей итерации). В подобных случаях, если это возможно, следует изменить алгоритм. На примере распараллеливания вложенных циклов удобно исследовать поведение программы во вложенных параллельных областях.

Пример 4.9. Программа содержит цикл, который невозможно корректно распараллелить

```
#include <stdio.h>
#include <math.h>
double f(double x){
    double result;
    код функции f
    return result;
}
int main(){
    double x1=0, x2=0, dx=1;
    //Этот цикл распараллелить нельзя!!!
    for (int i=0; dx<0.001; i++) {
        x2=f(x1); dx=fabs(x1-x2); x1=x2;
    }
}
```

Пример 4.10 демонстрирует распараллеливание двух вложенных циклов. Обратите внимание, что в примере до создания параллельных областей с помощью функций времени выполнения выставлены значения внутренних контрольных переменных, необходимых нам для предсказуемости поведения omp-программы во время выполнения. А именно, запрещено динамическое изменение количества потоков, разрешена вложенность параллельных областей, количество нитей, которое будет создаваться в каждой параллельной области, установлено равным 2, кроме того, зафиксировано статическое распределение итераций цикла по одной. Вторым циклом выводит на печать номер нити, выполняющей итерацию i первого цикла, и номер нити, выполняющей итерацию j второго цикла. Несмотря на то, что создание вложенных параллельных областей разрешено, директива предписывает распределение итераций только для внешнего цикла. Таким образом, каждая нить обрабатывает все пространство итераций внутреннего цикла для каждого значения i , т.е. в любом выводе значение переменной `num_th` будет равно значению, возвращаемому функцией `omp_get_thread_num()`.

Пример 4.10. Распределение итераций циклов

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char **argv){
    int j, i, n=3, m=2, num_th;
    omp_set_dynamic(0);
    omp_set_nested(1);
    omp_set_schedule(static,1);
    omp_set_num_threads(2);
    #pragma omp parallel
    #pragma omp for private(i,j)
    for (i=0; i<n; i++){
        num_th=omp_get_thread_num();
        for (j=0; j<m; j++)
            printf("%d - %d: (%d;%d)\n",
                num_th,omp_get_thread_num(),i,j);
    }
    return 0;
}
```

На рис. 4.3, а отображено все пространство итераций вложенного цикла и цветом отмечено, какая нить какие итерации выполняла.

Рассмотрим пример 4.11, в котором вложенный цикл примера 4.10 распараллелен. Поскольку вложенность параллельных областей разрешена, то каждая из двух нитей, выполняющих итерации внешнего цикла, по-

родит по две нити для выполнения итераций внутреннего цикла. Таким образом, цикл будет выполнен четырьмя нитями (рис. 4.3, б).

(0,0)	(0,1)	(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)	(1,0)	(1,1)
(2,0)	(2,1)	(2,0)	(2,1)	(2,0)	(2,1)
<i>a</i>		<i>б</i>		<i>в</i>	

Рис. 4.3. Распределение итераций цикла по нитям: *a* – распараллелен только внешний цикл (2 нити); *б* – распараллелены оба цикла (4 нити: вложение параллельных областей 2 по 2); *в* – распараллелено гнездо из двух плотновложенных циклов с помощью опции `collapse(2)` (2 нити)

Пример 4.11. Распределение итераций циклов. Вложение параллельных областей

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char **argv){
    int j, k, num_th;
    omp_set_dynamic(0);
    omp_set_nested(1);
    omp_set_schedule(static,1);
    omp_set_num_threads(2);
    #pragma omp parallel for private(k, j, num_th)
    for (k=0; k<=2; k++){
        num_th=omp_get_thread_num();
        #pragma omp parallel for private(j) shared(num_th)
        for (j=0; j<=1; j++){
            printf("%d - %d: (%d; %d)\n",
                num_th, omp_get_thread_num(), k, j);
        }
    }
    return 0;
}
```

Заметим, что цикл из примера 4.11 будет выполняться параллельно четырьмя нитями, однако это не значит, что он распараллелен более эффективно, чем цикл из примера 4.10, выполняемый двумя. Дело в том, что на каждой итерации внешнего цикла требуется дополнительное время на порождение нитей вложенной параллельной области, их синхронизацию и уничтожение. Если это время суммарно будет превосходить выигрыш от параллельного выполнения расчетов, то лучше не использовать вложение параллельных областей.

Для распараллеливания плотно вложенного гнезда циклов в примере 4.12 используется опция `collapse(2)`, сообщающая, что для выполнения двух циклов можно организовать общее пространство итераций и распределить работу между двумя нитями параллельной области. Заметим, что в этом случае итерации распределены наиболее равномерно (рис. 4.3, в).

Пример 4.12. Распределение итераций плотно вложенных циклов. Механизм collapse

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char **argv){
    int j, k;
    omp_set_dynamic(0);
    omp_set_nested(1);
    omp_set_num_threads(2);
    omp_set_schedule(static,1);
    #pragma omp parallel for collapse(2) private(k, j)
    for (k=0; k<=2; k++){
        for (j=0; j<=1; j++){
            printf("%d: (%d; %d)\n", omp_get_thread_num(), k, j);
            fflush(stdout);
        }
    }
    return 0;
}
```

Изменяя способ распределения итераций по нитям, проанализируйте выводы на консоль примеров 4.10–4.12.

Рассмотрим задачу, часто встречающуюся в теории игр.

Задача 4.1. Дан двумерный массив *a*, необходимо найти минимум *g_minmax* среди максимумов *amax[i]* всех строк двумерного массива.

В примере 4.13 в последовательный код решения этой задачи внесены минимальные изменения для параллельного выполнения цикла. Двумерный массив *a* заполняется случайными числами, максимум по каждой строке отдельно не хранится, а сразу накапливается в переменной *amax*. Внешний цикл вычисляет *g_minmax*.

Поскольку в теле как внутреннего, так и внешнего цикла есть запись в общие переменные *amax*, *g_minmax* несколькими нитями, то при выполнении кода примера неизбежно возникнет состояние гонок. Корректно решить эту проблему можно несколькими способами. В следующем пункте мы рассмотрим различные способы синхронизации нитей внутри параллельной области. В этом пункте приведем два других возможных корректных решения.

В примере 4.14 для устранения состояния «гонок» переменная *amax* объявляется локальной, глобальный минимум *G_minmax* вычисляется в два этапа. Сначала каждая нить насчитывает свой промежуточный минимум в элемент массива *g_minmax[i]*, совпадающий по номеру с номером нити. Таким образом, множества записей нитей в параллельной области не пересекаются, и состояние «гонок» не возникает. Затем в отдельном цикле определяется минимум *G_minmax* по массиву *g_minmax*. Поскольку этот массив

заведомо короткий (его размерность совпадает с количеством нитей), то цикл распараллеливать нет необходимости.

Пример 4.13. Определение минимального значения среди максимумов по строкам в двумерном массиве. Некорректное распараллеливание цикла

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <float.h>
#define N (int)10000
#define NUM_OF_TH (int)8
int main( int argc, char **argv ){
    int n=N, i, j;
    int size_th=(N>=NUM_OF_TH?NUM_OF_TH:1);
    double g_minmax=0, amax, *a;
    double omp_t1, omp_t2;
    //Динамическое выделение памяти и инициализация массива
    a=(double*)malloc(sizeof(double)*n*n);
    if(!a){printf ("error in a-malloc\n");exit(1);}
    srand(time(NULL));
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            a[i*n+j]=(double) (rand()%n);
    omp_set_num_threads(size_th);
    g_minmax=DBL_MAX;
    #pragma omp parallel for private(i,j)
    for (i=0;i<n;i++){
        amax=a[i*n];
        for (j=0;j<n;j++){
            if (amax<a[i*n+j]) amax=a[i*n+j];
        }
        if (g_minmax>amax) g_minmax=amax;
    }
    printf("g_minmax=%le\n",g_minmax);
    free(a);
    return 0;
}
```

Следует понимать, что хотя в основном цикле программы нет синхронизации, такая организация расчетов влечет за собой некоторые дополнительные временные затраты на выделение памяти под массив `g_minmax`, заполнение этого массива максимально возможными в типе `double` значениями `DBL_MAX`, поиск минимума `G_minmax`. Для определения, эффективно ли распараллеливание, в код введен замер времени выполнения программы. Обсуждение ускорения и эффективности примера 4.14 будет проведено ниже.

Другой способ решения проблемы «гонок» – использование операции редукции (`min`) для переменной при выходе из цикла. Пример 4.15 реализует этот подход. Отметим, что возможность использования операций `min` и `max` в опции `reduction` директив `omp parallel` и `omp for` для языков Си/Си++ появилась только в спецификации стандарта OpenMP v3.1.

Пример 4.14. Определение минимального значения среди максимумов по строкам в двумерном массиве. Без синхронизации нитей внутри цикла

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <float.h>
#define N (int)10
#define NUM_OF_TH (int)8
int main( int argc, char **argv ){
    int n=N, i, j;
    int size_th=(N>=NUM_OF_TH?NUM_OF_TH:1);
    double *g_minmax, G_minmax, amax, *a;
    double omp_t1, omp_t2;
    //см. соответствующий участок кода примера 4.13:
    //динамическое выделение памяти и инициализация массива
    ...
    omp_set_num_threads(size_th);
    omp_t1=omp_get_wtime();
    g_minmax=(double*)malloc(sizeof(double)* size_th);
    if(!g_minmax){printf ("error in amax-malloc\n");exit(1);}
    for (i=0;i<SIZE_TH;i++){
        g_minmax[i]=DBL_MAX;
        #pragma omp parallel shared(a,g_minmax)
                           private(i,j,amax,th_num)
        {
            th_num=omp_get_thread_num();
            #pragma omp for
            for (i=0;i<n;i++){
                amax=a[i*n];
                for (j=0;j<n;j++){
                    if (amax<a[i*n+j]) amax=a[i*n+j];
                    if (g_minmax[th_num]>amax) g_minmax[th_num]=amax;
                }
            }
        }
        G_minmax=DBL_MAX;
        for (i=0;i<SIZE_TH;i++){
            if (G_minmax>g_minmax[i]) G_minmax=g_minmax[i];
        }
        omp_t2=omp_get_wtime();
        printf("g_minmax=%le\n",g_minmax);
        printf("%le\n",omp_t2-omp_t1);
        free(a);
        return 0;
    }
```

Пример 4.15. Определение минимального значения среди максимумов по строкам в двумерном массиве. Механизм редукции

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <float.h>
#define N (int)10
#define NUM_OF_TH (int)8
int main( int argc, char **argv ){
    int n=N, i, j;
    double g_maxmin=0, g_minmax=0, amax, amin, *a;
    double omp_t1, omp_t2;
    //см. соответствующий участок кода Примера 4.13:
    //динамическое выделение памяти и инициализация массива
    ...
    omp_set_num_threads(SIZE_TH);
    omp_t1=omp_get_wtime();
    g_minmax=DBL_MAX;
    #pragma omp parallel for shared(a) private(amax,i,j) \
                        reduction(min:g_minmax)

        for (i=0;i<n;i++){
            amax=a[i*n];
            for (j=0;j<n;j++)
                if (amax<a[i*n+j]) amax=a[i*n+j];
            if (g_minmax>amax) g_minmax=amax;
        }
    omp_t2=omp_get_wtime();
    printf("g_minmax=%le\n",g_minmax);
    printf("%le\n",omp_t2-omp_t1);
    free(a);
    return 0;
}
```

Если некоторый блок операторов в распараллеленном цикле следует обработать в том порядке, в каком он обрабатывался бы при последовательном выполнении, то в директиве `omp for` указывается опция `ordered`, а структурный блок оформляется директивой

```
#pragma omp ordered
    структурный блок
```

В теле цикла может встречаться только одна директива `omp ordered`.

Директива **task**

Для реализации подхода декомпозиции по задачам в OpenMP предусмотрена специальная директива

```
#pragma omp task [опция[, опция]...]
    структурный блок
```

позволяющая выделить отдельную независимую задачу, код которой определен в структурном блоке. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией. В директиве `omp task` могут быть определены следующие опции:

- `private`, `firstprivate`, `shared`, `default(shared|none)` – эти опции определяются точно так же, как и в директиве `parallel`;
- `if(выражение)` – если значение выражения отлично от нуля, то новая задача будет порождена, иначе код структурного блока будет немедленно выполнен текущей нитью (пример 4.16);
- `untied` – если выполнение задачи прервано, то ее может продолжить выполнять любая нить в параллельной области. Если опция не указана, то продолжить выполнение отложенной задачи может только нить, которая изначально взяла ее на выполнение;
- `final(выражение)` – если значение выражения отлично от нуля, то новая задача будет порождена как конечная (`final`), т. е. все вложенные задачи также будут конечные и должны безотлагательно выполняться, причем той же самой нитью, что и родительская (пример 4.16). При истинности выражения в `final` опция `untied`, если она присутствует в этой же директиве, игнорируется. Использование опции полезно для ограничения порождения новых нитей при параллельной рекурсии, поскольку глубокая рекурсия может исчерпать аппаратные и/или программные ограничения на порождение новых нитей;
- `mergeable` – указывает, что если порожденная задача «погружена» в родительскую, то она будет использовать одни и те же данные, включая IS-переменные. Иначе поведение определяется реализацией.

Функция

```
int omp_in_final(void);
```

возвращает 1, если выполняющийся код принадлежит конечной задаче, и 0 в противном случае.

Для управления задачами в OpenMP определены две директивы.

```
#pragma omp taskyield
```

Эта директива сообщает, что задачу можно в этой точке прервать для выполнения других задач. Если задача порождена с опцией `untied`, то ее может затем продолжить выполнять любая нить группы.

```
#pragma omp taskwait
```

Нить, выполнившая эту директиву, будет приостановлена до тех пор, пока не будут завершены все ранее запущенные данной нитью независимые задачи.

Пример 4.16. Фрагмент кода для иллюстрации понятия конечная (`final`) задача

```
void foo2(void); //функция, определяющая код некоторой задачи
void foo1() {
    int i;
    #pragma omp parallel num_threads(5)
    {
        //Задача будет выполнена немедленно текущей нитью
        #pragma omp task if(0)
        {
            //Будет порождена обычная задача для выполнения цикла
            #pragma omp task
            for (i=0; i<4; i++){
                //Будет порождена обычная задача
                //для выполнения функции foo2()
                #pragma omp task
                foo2() ;
            }
        }
        #pragma omp task final(1)
        //Будет порождена конечная задача:
        //все вложенные задачи будут выполняться
        //немедленно и той же нитью, что и порожденная
        {
            //Будет порождена «погруженная» задача
            #pragma omp task
            for (i=0; i<4; i++){
                //И эта задача будет «погруженная»
                #pragma omp task
                foo2() ;
            }
        }
    }
}
```

Заметим, что обе директивы являются примером небольшого количества *автономных* директив OpenMP. Такие директивы не связаны с каким-либо блоком кода программы и предназначены для выполнения сами по себе. Автономная директива, если она должна выполняться в теле цикла или условного оператора, всегда стоит в операторных скобках.

Иллюстрация понятия конечной нити приведена в примере 4.16, где в первый раз выполнение функции `foo2()`, скорее всего, будет поручено четырем нитям, во второй раз все четыре вызова будут выполнены одной нитью.

Пример 4.17 демонстрирует использование концепции явного формирования задач для вычисления произведения матриц с помощью парадигмы «портфель задач». Задачей является вычисление одного элемента результирующей матрицы. Код задачи описан в функции `element`. В функции `main()` сначала декларируются и инициализируются матрицы, затем выставляются необходимые значения внутренних контрольных переменных, а в параллельной области в цикле формируются задачи, вызывающие функцию `element`.

Обратите внимание, что пул задач в примере 4.17 формируется одной нитью в области действия директивы `omp single`. Когда все задачи будут сформированы, они распределятся между нитями параллельной области для выполнения. Порядок выполнения задач и номер нити, которой будет назначена конкретная задача, заранее не определены. В конце параллельной области будет выполнена неявная барьерная синхронизация, после чего на печать будет выведена результирующая матрица. Код функции `prn_2d()` совпадает с примером 4.4.

Пример 4.17. Вычисление произведения матриц. Реализация портфеля задач с помощью механизма `task`. Каждая задача в портфеле: вычисление одного элемента результирующей матрицы

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//Размерность квадратной матрицы в одном направлении
const int n = 1000;
//Функция печати из Примера 4.4
void prn_2d (int* array, int nn, int mm);
//Задача: функция умножения строки матрицы a
//на столбец матрицы b
void element(int ii, int jj, int* a, int* b,
             int* current_element){
    int kk;
    int num_th=omp_get_thread_num();
    printf("num_th = %d\n",num_th);
    *current_element = 0.0;
    for (kk=0 ;kk<n;kk++)
        *current_element+= a[ii*n+kk] * b[kk*n+jj];
}
int main(int argc, char **argv){
    int i,j, *a,*b,*c;
```

Пример 4.17 (окончание). Вычисление произведения матриц. Реализация портфеля задач с помощью механизма task. Каждая задача в портфеле: вычисление одного элемента результирующей матрицы

```
a=(int*)malloc(sizeof(int)*n*n);
if(!a){printf ("error in a-malloc\n");exit(1);}
b=(int*)malloc(sizeof(int)*n*n);
if(!b){printf ("error in b-malloc\n");exit(1);}
c=(int*)malloc(sizeof(int)*n*n);
if(!c){printf ("error in amin-malloc\n");exit(1);}
//Инициализировать матрицы a и b каким-либо способом
...
printf("A\n"); prn_2d(a,n,n);
printf("B\n"); prn_2d(b,n,n);
printf("C\n"); prn_2d(c,n,n);
omp_set_dynamic(0);
omp_set_nested(1);
omp_set_num_threads(8);
#pragma omp parallel shared(a,b,c,n) private(i,j)
{
    #pragma omp single
    {
        for (i=0;i<n;i++)
            for (j=0;j<n;j++)
                //По умолчанию a,b,c,n - shared, i,j - firstprivate
                #pragma omp task
                element(i,j,a,b,&c[i*n+j]);
    }
}
printf("C\n"); prn_2d(c,n,n);
return 0;
}
```

Как уже отмечалось, механизм `omp task` удобно использовать для организации параллельной рекурсии. В примере 4.18 продемонстрирована организация рекурсии для вычисления интеграла с удовлетворительной точностью с помощью адаптивной квадратуры (задача 1.2). Метод вычисления интеграла подробно обсуждался в гл. 1. В примере 4.18 рекурсия применяется для вычисления значения числа π по формуле

$$\pi = 4 \int_0^1 \sqrt{1-x^2} dx.$$

Для вычисления интеграла отрезок рекурсивно делится

пополам, и если точность вычисления интеграла не достигнута, то порождается две ветви рекурсии. Таким образом, программа осуществляет прямой и обратный обход несбалансированного, в общем случае, бинарного дерева. Поскольку интеграл находится как сумма интегралов по отрезкам,

то значения вычисленных слагаемых должны быть доступны всем задачам каждой ветви рекурсии, поэтому при порождении задачи используется опция `shared`. Для корректной работы параллельной рекурсии необходимо, чтобы правые и левые ветви были выполнены до обработки текущего узла, поэтому используется директива `omp taskwait`.

Пример 4.18. Адаптивная квадратура на основе выделения независимых задач. Организация параллельной рекурсии

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define EPS (double)1.e-15
double f(double x){
    return sqrt(1. - x*x);
}
double q_integral (double left, double right,
    double f_left, double f_right, double int_current){
    double mid = (left + right) / 2;
    double f_mid = f(mid);
    //левый отрезок
    double l_integral=(f_left+f_mid)*(mid-left)/2;
    //правый отрезок
    double r_integral=(f_mid+f_right)*(right-mid)/2;
    if (fabs((l_integral +r_integral)- int_current) > EPS){
        // рекурсия
        #pragma omp task shared(l_integral)
        l_integral = q_integral(left, mid,
                                f_left, f_mid, l_integral);
        #pragma omp task shared(r_integral)
        r_integral = q_integral(mid, right,
                                f_mid, f_right, r_integral);
    }
    #pragma omp taskwait
    return (l_integral + r_integral);
}
int main(int argc, char **argv){
    double a=0, b=1, area=0;
    omp_set_dynamic(0);
    omp_set_nested(1);
    omp_set_num_threads(8);
    #pragma omp parallel
    {
        area=q_integral(a,b,f(a),f(b), (f(a)+f(b)) * (b-a) / 2);
    }
    printf ("pi = %le", 4.0*area);
    return 0;
}
```

4.6. Директивы синхронизации

При одновременном выполнении нескольких нитей часто возникает необходимость их синхронизации. Рассмотрим механизмы, предоставляемые для этого OpenMP.

Директива `omp barrier`

Обязательная неявная барьерная синхронизация выполняется в конце каждой параллельной области для всех сопоставленных с ней нитей. Кроме того, неявно синхронизация барьером выполняется в конце области действия директив `omp for`, `omp single` и `omp sections`, однако в этом случае барьер может быть исключен опцией `nowait`.

В OpenMP можно определить явную барьерную синхронизацию автономной директивой

```
#pragma omp barrier
```

Выполняющая эту директиву нить блокируется и ждет, пока все нити группы в текущей параллельной области не выполнят эту же директиву, после чего все нити получают возможность продолжить работу. Для разблокировки также необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (task).

Механизм замков `locks`

Одним из способов синхронизации нитей параллельной области является механизм замков (locks). Замок – это переменная специального типа, работа с которой осуществляется только соответствующими функциями времени выполнения. Алгоритм работы с замками повторяет работу с мьютексами (например, переменными типа `pthread_mutex_t` библиотеки Pthread и объектом ядра «мьютекс» в WinAPI).

Замок `lock` может пребывать только в одном из трех состояний: *неинициализированном*, *свободном* или *занятом*. Свободный замок может быть захвачен некоторой нитью, при этом он переходит в занятое состояние. Нить, захватившая замок, и только она, может его освободить, т. е. вернуть замок в свободное состояние.

Есть два типа замков: *простые* и *рекурсивные*. Простой замок может быть захвачен перед освобождением только однажды, а рекурсивный – многократно. Для рекурсивного замка введено понятие коэффициента захвата (nesting count). Изначально коэффициент захвата равен нулю, при каждом следующем захвате он увеличивается на единицу, а при каждом освобождении – уменьшается на единицу. Рекурсивный замок считается свободным, если его коэффициент захвата равен нулю.

Для простых и рекурсивных замков в файле `omp.h` определены специальные типы `omp_lock_t` и `omp_nest_lock_t` соответственно. Ниже перечислены функции времени выполнения определенные в OpenMP для работы с замками. Первая функция из каждой пары соответствует работе с простым замком, вторая – с рекурсивным.

```
void omp_init_lock(omp_lock_t *lock);  
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

Инициализация свободного замка. Для рекурсивного замка коэффициент захвата устанавливается в 0.

```
void omp_destroy_lock(omp_lock_t *lock);  
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

Функции переводят замок в неинициализированное состояние.

```
void omp_set_lock(omp_lock_t *lock);
```

Функция атомарно дожидается освобождения простого замка и переводит его в занятое состояние.

```
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

Функция, вызванная для рекурсивного замка первый раз (коэффициент захвата равен нулю), атомарно дожидается освобождения замка и переводит его в занятое состояние, а все последующие вызовы без задержки увеличивают на единицу коэффициент захвата.

```
void omp_unset_lock(omp_lock_t *lock);
```

Если простой замок был захвачен нитью, вызвавшей функцию, то замок переводится в свободное состояние, иначе функция отработывает без ошибок, и никаких действий не происходит.

```
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

Если рекурсивный замок был захвачен нитью, вызвавшей функцию, то коэффициент захвата уменьшается на единицу, в случае, если после этого коэффициент захвата становится равен нулю, замок переводится в свободное состояние. Если функцию вызвала нить, которая не занимала замок, то функция отработывает без ошибок, и никаких действий не происходит.

Если при освобождении замка существует хотя бы одна нить, ожидающая его захвата, то замок будет немедленно захвачен, причем порядок захвата при ожидании несколькими нитями заранее не определен.

```
int omp_test_lock(omp_lock_t *lock);  
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1, а для рекурсивного замка – новый коэффициент захвата. Если замок захватить не удалось, в обоих случаях возвращается 0.

Пример 4.19. Определение минимального значения среди максимумов по строкам в двумерном массиве. Максимальный параллелизм. Механизм замков

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <float.h>
#define N (int)10
#define SIZE_TH (int)8
int main( int argc, char **argv ){
    int n=N, i, j, size_th=(N>=NUM_OF_TH?NUM_OF_TH:1);
    double g_maxmin=0, g_minmax=0, amax, amin, *a;
    double omp_t1, omp_t2;
    omp_lock_t lock;
    omp_init_lock(&lock);
    //Динамическое выделение памяти и инициализация массива
    a=(double*)malloc(sizeof(double)*n*n);
    if(!a){printf ("error in a-malloc\n");exit(1);}
    srand(time(NULL));
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            a[i*n+j]=(double) (rand()%n);
    omp_set_num_threads(size_th);
    omp_set_nested(1);
    omp_t1=omp_get_wtime();
    g_minmax=DBL_MAX;
    #pragma omp parallel for shared(a,g_minmax) \
                        private(amax,i,j)
    for (i=0;i<n;i++){
        amax=a[i*n];
        #pragma omp parallel for shared(a,amax) private(j)
        for (j=0;j<n;j++){
            omp_set_lock(&lock);
            if (amax<a[i*n+j]) amax=a[i*n+j];
            omp_unset_lock(&lock);
        }
        omp_set_lock(&lock);
        if (g_minmax>amax) g_minmax=amax;
        omp_unset_lock(&lock);
    }
    omp_t2=omp_get_wtime();
    printf("g_minmax=%le\n",g_minmax);
    printf("%le\n",omp_t2-omp_t1);
    omp_destroy_lock(&lock);
    free(a);
    return 0;
}
```

Следует понимать, что на программисте лежит ответственность за исключение мертвых блокировок при использовании замков – все замки должны быть сначала установлены, а затем освобождены одной и той же нитью.

Пример 4.20. Определение минимального значения среди максимумов по строкам в двумерном массиве. Оптимизация кода: отказ от вложенных параллельных областей

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <float.h>
#define N (int)10
#define SIZE_TH (int)8
int main( int argc, char **argv ){
    int n=N, i, j;
    int size_th=(N>=NUM_OF_TH?NUM_OF_TH:1);
    double g_maxmin=0, g_minmax=0, amax, amin, *a;
    double omp_t1, omp_t2;
    omp_lock_t lock;
    omp_init_lock(&lock);
    //см. соответствующий участок кода примера 4.18:
    //динамическое выделение памяти и инициализация массива
    ...
    omp_set_num_threads(size_th);
    omp_t1=omp_get_wtime();
    g_minmax=DBL_MAX;
    #pragma omp parallel for shared(a,g_minmax) \
                        private(amax,i,j)

    for (i=0;i<n;i++){
        amax=a[i*n];
        for (j=0;j<n;j++){
            if (amax<a[i*n+j]) amax=a[i*n+j];
            omp_set_lock(&lock);
            if (g_minmax>amax) g_minmax=amax;
            omp_unset_lock(&lock);
        }
    }
    omp_t2=omp_get_wtime();
    printf("g_minmax=%le\n",g_minmax);
    printf("%le\n",omp_t2-omp_t1);
    omp_destroy_lock(&lock);
    free(a);
    return 0;
}
```

Использование замков является наиболее гибким механизмом синхронизации нитей, поскольку с помощью замков можно реализовать все

остальные варианты синхронизации. Однако с точки зрения накладных расходов работа с замками – достаточно дорогой механизм (см. табл. 4.3).

Пример 4.21. Определение минимального значения среди максимумов по строкам в двумерном массиве. Оптимизация критического кода: предварительная проверка

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <float.h>
#define N (int)10
#define SIZE_TH (int)8
int main( int argc, char **argv ){
    int n=N, i, j;
    int size_th=(N>=NUM_OF_TH?NUM_OF_TH:1);
    double g_maxmin=0, g_minmax=0, amax, amin, *a;
    double omp_t1, omp_t2;
    omp_lock_t lock;
    omp_init_lock(&lock);
    / см. соответствующий участок кода примера 4.18:
    //динамическое выделение памяти и инициализация массива
    ...
    omp_set_num_threads(size_th);
    omp_t1=omp_get_wtime();
    g_minmax=DBL_MAX;
    #pragma omp parallel for shared(a,g_minmax) \
                        private(amax,i,j)

    for (i=0;i<n;i++){
        amax=a[i*n];
        for (j=0;j<n;j++)
            if (amax<a[i*n+j]) amax=a[i*n+j];
        if (g_minmax>amax){
            omp_set_lock(&lock);
            if (g_minmax>amax) g_minmax=amax;
            omp_unset_lock(&lock);
        }
    }
    omp_t2=omp_get_wtime();
    printf("g_minmax=%le\n",g_minmax);
    printf("%le\n",omp_t2-omp_t1);
    omp_destroy_lock(&lock);
    free(a);
    return 0;
}
```

Рассмотрим задачу 4.1 поиска минимума среди максимумов строк двумерного массива. Варианты решения этой задачи без явной синхрони-

зации были рассмотрены в примерах 4.14, 4.15. В примере 4.19 приведен код программы с синхронизацией внутри цикла для обновления общих переменных с помощью замков. Обратите внимание на то, какие переменные объявлены общими, а какие – локальными в каждой из вложенных параллельных областей.

Поскольку переменная `amax` является общей для вложенной секции, то ее возможное обновление защищено замком. Аналогично замком защищена возможная запись в переменную `g_minmax`. Подумайте, можно ли защитить обновление `amax` и `g_minmax` разными замками.

Применение вложенных параллельных областей в примере 4.19 не эффективно. Это связано не только с накладными расходами на порождение лишних нитей, но и с расходами на дополнительную защиту общей переменной `amax`. Если распараллеливать только внешний цикл, то одну критическую секцию можно убрать (пример 4.20).

Запись в общую переменную `g_minmax` происходит не всегда, а только если выполняется условие минимальности `amax`. Дальнейшая оптимизация кода (пример 4.21) связана с предварительной незащищенной проверкой выполнения условия. Защищенная запись в `g_minmax` будет производиться только тогда, когда условие незащищенной проверки истинно. Повторная проверка перед записью необходима, поскольку нет гарантии, что общая переменная `g_minmax` будет все еще оставаться больше локальной переменной `amax`. Значение `g_minmax` может быть изменено другой нитью между первой проверкой и установкой замка.

Директива `omp critical`

В OpenMP участки кода, которые должны выполняться с взаимным исключением, можно выделить как критическую секцию с помощью директивы

```
#pragma omp critical [(имя_критической_секции)]  
структурный блок
```

В каждый момент времени в критической секции может находиться не более одной нити. Нить, встретившая директиву, проверяет, выполняет ли какая-либо другая нить критическую секцию с таким же именем. Если критическая секция занята, то нить приостанавливает свою работу, в противном случае – занимает критическую секцию и выполняет код структурного блока. Как только работавшая нить покинет критическую секцию, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию ожидало несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и то же имя, рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

Пример 4.22. Определение минимального значения среди максимумов по строкам в двумерном массиве. Механизм критической секции

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <float.h>
#define N (int)10
#define SIZE_TH (int)8
int main( int argc, char **argv ){
    int n=N, i, j;
    double g_maxmin=0, g_minmax=0, amax, amin, *a;
    double omp_t1, omp_t2;
    //см. соответствующий участок кода примера 4.19:
    //динамическое выделение памяти и инициализация массива
    ...
    omp_set_num_threads(SIZE_TH);
    omp_t1=omp_get_wtime();
    g_minmax=DBL_MAX;
    #pragma omp parallel for shared(a,g_minmax) \
        private(amax,i,j)

    for (i=0;i<n;i++){
        amax=a[i*n];
        for (j=0;j<n;j++)
            if (amax<a[i*n+j]) amax=a[i*n+j];
        if (g_minmax>amax){
            #pragma omp critical (min)
            if (g_minmax>amax) g_minmax=amax;
        }
    }
    omp_t2=omp_get_wtime();
    printf("g_minmax=%le\n",g_minmax);
    printf("%le\n",omp_t2-omp_t1);
    free(a);
    return 0;
}
```

В примере 4.22 задача 4.1 поиска минимума среди максимумов строк двумерного массива решена с помощью механизма критической секции. В примере приведен только участок кода, отличающийся от программы

примера 4.19. Обратите внимание, какие переменные объявлены общими, а какие локальными для параллельной области. Внутренний цикл читает и, возможно, записывает локальную переменную `amax`, а обновление общей переменной `g_minmax` безопасно, поскольку защищено критической секцией.

Директива `omp atomic`

Директива `omp atomic` в одной из следующих форм:

```
#pragma omp atomic [read | write | update | capture]
выражение
```

или

```
#pragma omp atomic capture
структурный блок
```

позволяет гарантировать неделимое выполнение относительно переменных `x` и `v` следующего за ней оператора или структурного блока в объеме, определенном одной из следующих опций:

- `read` – атомарное чтение из `x`;
- `write` – атомарная запись в `x`;
- `update` – атомарные чтение и/или запись `x` (применяется по умолчанию);
- `capture` – атомарный захват начального `x` и финального `v` значений.

Формат выражения, на которое действует директива, должен быть следующий:

- для опции `read`: `v=x`;
- для опции `write`: `x=expr`;
- для опции `update` или когда опция не указана: `x++` | `x--` | `++x` | `--x` | `x binop = expr` | `x=x binop expr`;
- для опции `capture`: `v=x++` | `v=x--` | `v=++x` | `v=--x` | `v=x binop=expr`.

Формат структурного блока, на который действует директива, должен быть одним из следующих:

```
{v=x; x binop=expr;} | {x binop=expr; v=x;} | {v=x; x=x binop
expr;} | {x=x binop expr; v=x;} | {v=x; x++;} | {v=x; ++x;} |
{++x; v=x;} | {x++; v=x;} | {v=x; x--;} | {v=x; --x;} | {--x;
v=x;} | {x--; v=x;}.
```

Здесь `x` и `v` – переменные скалярного типа, через `binop` обозначена бинарная операция, допустимая в Си, `expr` – скалярное выражение, допус-

тимое в Си, в том числе и вызов функции, которая не имеет побочных эффектов для x и v .

Применение директивы `omp atomic` для атомарного изменения значений переменных x и v несколькими нитями выгодно по сравнению с защитой с помощью замков или директивой `omp critical`, поскольку все выражения, участвующие в определении x , могут вычисляться параллельно. Кроме того, если защищаемые `omp atomic` области памяти не пересекаются для разных нитей, то операции выполняются без излишних задержек.

В примере 4.23 обновление в (1) разных $x[j]$ может идти параллельно. Обратите внимание, что директива `omp atomic` действует непосредственно на оператор (1), и обновление $y[i]$ в (2) происходит опасно, без синхронизации.

Пример 4.23. Механизм `omp atomic`

```
#define N1 1000
#define N2 10000
double foo1(int i){return 1.0 * i;}
double foo2(int i){return 2.0 * i;}
void atomic_example(double *x,double *y,int *index,int n){
    int i;
    #pragma omp parallel for shared(x, y, index, n)
    for (i=0; i<n; i++) {
        #pragma omp atomic update
        x[index[i]] += foo1(i); // (1)
        y[i] += foo2(i); // (2)
    }
}

int main(){
    float x[N1],y[N2];
    int i, index[N2];
    for (i = 0; i < N2; i++) {
        index[i] = i % N1; y[i]=0.0;
    }
    for (i = 0; i < N1; i++)
        x[i] = 0.0;
    atomic_example(x, y, index, N2);
    return 0;
}
```

Директива `omp flush`

Как уже отмечалось в начале главы, в OpenMP используется ослабленная модель согласования общей памяти (*relaxed-consistency shared-memory model*), которая подразумевает следующее поведение параллельной программы:

- все нити видят одну и ту же последовательность записей в общую память (разрешаются запаздывания для чтения);
- каждая нить имеет собственное временное представление памяти (*temporary view of the memory*);
- допустима временная несогласованность между представлением памяти нити и общей памятью.

Такая модель памяти таит в себе много неочевидных ловушек. Если в последовательной программе нить сначала записывает новое значение переменной, а затем использует его по чтению, то аппаратúra и компилятор гарантируют, что использовать мы будем новое значение. Если же мы имеем дело с конкурентным выполнением записи в общую переменную одной нитью, а использование этой переменной другой нитью, то возможно временное хранение нового значения в локальном представлении памяти первой нити и чтение его старого значения из общей памяти второй нитью.

Рассмотрим псевдокод примера 4.24. Согласно нотации, введенной в гл. 1, в угловых скобках указаны атомарные операции. Цель синхронизации нитей в псевдокоде – предотвратить выполнение кода защищаемой секции одновременно двумя нитями.

Разумно ожидать несколько возможных последовательностей выполнения операторов (1)–(4) (табл. 4.3). Из таблицы видно, что код защищаемой секции либо будет выполнен одной из нитей (выделенные строки), либо не будет выполнен вовсе. Однако наличие временного представления памяти у каждой нити запутывает их совместное поведение! Рассмотрим, например, первую в табл. 4.3 последовательность выполнения смеси операторов (1)–(4). В OpenMP-программе возможно, что первая нить к моменту выполнения оператора (4) все еще не обновит единицей значение переменной *b* в общей памяти, а вторая нить будет использовать старое, нулевое, значение *b*. В этом случае код защищаемой секции выполнят обе нити.

Пример 4.24. Псевдокод: защищаемая секция должна быть выполнена не более чем одной нитью. Однако ослабленная модель согласования общей памяти этого не гарантирует!

a=0; b=0;

Псевдокод thread 1

<b = 1>; // (1)

<tmp1 = a>; // (2)

if (tmp1 == 0) then {

код защищаемой секции

}

Псевдокод thread 2

<a = 1>; // (3)

<tmp2 = b>; // (4)

if (tmp2 == 0) then {

код защищаемой секции

}

Для принудительного согласования временной и общей памяти нити в OpenMP используется автономная директива

`#pragma omp flush(список_переменных)`

Директива приостанавливает выполнение нити до тех пор, пока не завершатся все незавершенные к этому времени обмены с общей памятью, касающиеся перечисленных в списке переменных. Если списки переменных для синхронизации в двух директивах `omp flush` пересекаются, то запрещено переупорядочивание соответствующих согласований. Если список переменных отсутствует, то перед выполнением следующего за директивой оператора программы должны завершаться все незавершенные обмены с общей памятью.

Таблица 4.3
Возможные последовательности
выполнения операторов (1) – (4)
в псевдокоде 4.23

Порядок				tmp1	tmp2
1	2	3	4	0	1
1	3	2	4	1	1
1	3	4	2	1	1
3	4	1	2	1	0
3	1	2	4	1	1
3	1	4	2	1	1

Изменим псевдокод примера 4.24, добавив в него точки синхронизации памяти, как показано в примере 4.25.

Пример 4.25. Псевдокод: защищаемая секция должна быть выполнена не более чем одной нитью. Однако ослабленная модель согласования общей памяти этого не гарантирует!

`a=0;b=0;`

Псевдокод thread 1

`<b = 1>; // (1)`

`#pragma omp flush (b)`

`//(s1)`

`#pragma omp flush (a)`

`//(s2)`

`<tmp1 = a>; // (2)`

`if (tmp1 == 0) then {`

код защищаемой секции

`}`

Псевдокод thread 2

`<a = 1>; // (3)`

`#pragma omp flush (a) //(s3)`

`#pragma omp flush (b) //(s4)`

`<tmp2 = b>; // (4)`

`if (tmp2 == 0) then {`

код защищаемой секции

`}`

Однако и этот псевдокод содержит ошибку. Поскольку списки переменных для согласования в (s1)–(s2) или (s3)–(s4) не пересекаются, то возможно переупорядочивание компилятором этих директив внутри нити. Например, первая нить после оператора (1) не использует нигде значение переменной `b`; следовательно, компилятор может перенести выполнение операции согласования (s1) в конец кода нити. Аналогично во второй нити согласование (s3) может быть выполнено после кода защищенной секции.

В обоих случаях защищенная секция будет выполнена обеими нитями. Чтобы предотвратить переупорядочивание, необходимо объединить списки для синхронизации в обеих нитях (пример 4.26).

Неявное принудительное согласование временной и общей памяти нити для всех общих переменных параллельной области происходит в следующих случаях:

- в случае барьерной синхронизации;
- при входе и выходе из параллельной области, критической секции, области действия директивы `omp ordered`;
- при выполнении синхронизации замками (`omp_set_lock` и `omp_unset_lock`, `omp_test_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock` и `omp_test_nest_lock`);
- немедленно перед приостановкой и немедленно после возобновления выполнения задачи.

Пример 4.26. Псевдокод: защищаемая секция должна быть выполнена не более чем одной нитью. Правильная реализация в рамках ослабленной модели согласования общей памяти

`a=0;b=0;`

Псевдокод thread 1

```
<b = 1>; // (1)
#pragma omp flush (a,b)
<tmp1 = a>; // (2)
if (tmp1 == 0) then {
    код защищаемой секции
}
```

Псевдокод thread 2

```
<a = 1>; // (3)
#pragma omp flush (a,b)
<tmp2 = b>; // (4)
if (tmp2 == 0) then {
    код защищаемой секции
}
```

В примере 4.27 для демонстрации работы директив `omp atomic` и `omp flush` реализован алгоритм поликлиники с организацией очереди входа нитей в критическую секцию (см. гл. 2).

Для организации очереди объявлена глобальная структура `lock`, первое поле (`ticket_number`) которой используется для получения номера при постановке нити в очередь, а второе поле (`turn`) необходимо для разрешения выполнения критического участка кода и сигнализации об его окончании. Оба поля инициализированы нулями. В основном потоке в параллельной области инициализировано несколько задач, которые необходимо выполнить с взаимным исключением. Содержательная часть кода каждой задачи описана функцией `work_smb()` (в нашем примере сама функция не уточняется).

Протоколы входа и выхода для критической секции опираются на реализацию атомарного доступа к полям структуры `lock`. Функция `atomic_read()` позволяет атомарно читать поля структуры, а функция

`fetch_and_add()` атомарно увеличивает на единицу значение поля структуры, возвращая его старое значение.

Пример 4.27. Механизм `omp atomic capture`: реализация алгоритма поликлиники для работы в критической секции

```
//Структура, которая используется функцией fetch_and_add()
//для реализации взаимного исключения
struct locktype{
    int ticket_number; // неделимое получение номера очереди
    int turn; // разрешение на вход и сигнал выхода из КС
};
//Глобальная переменная синхронизации
struct locktype lock{0,0};
//Атомарное чтение значения *p
int atomic_read(const int *p){
    int value;
    #pragma omp atomic read
    value = *p;
    return value;
}
//Атомарное увеличение значения аргумента *p на единицу
//Функция возвращает входное (старое) значение аргумента *p
int fetch_and_add(int *p) {
    int old;
    #pragma omp atomic capture
    { old = *p; (*p)++; }
    return old;
}
//Функция, которая должна выполняться со взаимным исключением
void work_smb();
void do_locked_work(struct locktype *lock){
    //Получить номер очереди входа в критическую секцию
    int myturn = fetch_and_add(&lock->ticket_number);
    //Протокол входа:
    //дождаться пока разрешение на вход lock->turn
    //не станет равно номеру в очереди
    while (atomic_read(&lock->turn) != myturn);
    //Функция work() выполняется с взаимным исключением
    //директива omp flush обновляет переменные
    #pragma omp flush
    work_smb();
    #pragma omp flush
    //Протокол выхода из КС
    fetch_and_add(&lock->turn) ;
}
```

Пример 4.27 (окончание). Механизм `omp atomic capture`: реализация алгоритма поликлиники для работы в критической секции

```
int main() {
    omp_set_dynamic(0);
    #pragma omp parallel private(i)
    {
        #pragma omp single
        {
            for (i=0; i<10; i++)
                #pragma omp task
                do_locked_work(&lock);
        }
    }
    return 0;
}
```

Функция `do_locked_work()` оборачивает вызов `work_smb()` протоколом входа в критическую секцию и протоколом выхода. Протокол входа реализует алгоритм поликлиники, обеспечивающий свойство живучести. Сначала каждая задача встает в очередь, выполняя функцию `fetch_and_add()`, атомарно увеличивая на единицу поле `ticket_number` структуры `lock` и запоминая свой номер в локальной переменной `myturn`. Затем выполняется активное ожидание, при котором задача все время сравнивает свой номер очереди с текущим, который хранится в поле `turn` структуры `lock`. Как только очередь подошла, задача синхронизирует временную память нити с общей памятью и вызывает `work_smb()`. После выполнения работы задача вновь синхронизирует память и выполняет протокол выхода, атомарно увеличивая на единицу поле `turn` структуры `lock`.

4.7. Переменные среды и функции времени выполнения

В начале главы (см. п. 4.3) были рассмотрены основные составляющие, формирующие среду выполнения OpenMP-программы: переменные среды, внутренние контрольные переменные, функции времени выполнения.

Рассмотрим несколько примеров, иллюстрирующих поведение среды выполнения.

Код примера 4.28 демонстрирует область действия некоторых полезных IS-переменных. Прежде всего, в программе запрещено динамическое изменение количества нитей в параллельной области. Далее устанавливаются две переменные, связанные с вложенностью параллельных областей: разрешается до четырех уровней вложенности. Все эти IS-переменные яв-

ляются глобальными и могут изменяться только вне параллельной области. Область действия IS-переменной, отвечающей за количество нитей, которое будет создано в ближайшей параллельной области, – параллельная область. В примере 4.28 значение этой переменной с помощью функции `omp_set_num_threads()` изменяется три раза. В первой параллельной области будет создано две нити. Затем каждая из этих нитей породит вложенные параллельные секции из трех нитей. В этих вложенных параллельных областях значение IS-переменной, отвечающей за количество нитей в следующей параллельной секции, опять изменится. Об этом можно судить по выводу, помеченному OUT1:

```
OUT1, 1: max_act_lev=4, num_thds=3, max_thds=4
OUT1, 0: max_act_lev=4, num_thds=3, max_thds=4
```

Пример 4.28. Область действия IS-переменных

```
#include <omp.h>
#include <stdio.h>
int main (void){
    omp_set_dynamic(0);
    omp_set_nested(1);
    omp_set_max_active_levels(4);
    omp_set_num_threads(2);
    #pragma omp parallel
    {
        omp_set_num_threads(3);
        #pragma omp parallel
        {
            omp_set_num_threads(4);
            #pragma omp single
            {
                printf ("OUT1, thread %d:
                        max_act_lev=%d, num_thds=%d, max_thds=%d\n",
                        omp_get_ancestor_thread_num(1),
                        omp_get_max_active_levels(),
                        omp_get_num_threads(), omp_get_max_threads());
            }
        }
        #pragma omp barrier
        #pragma omp single
        {
            printf("OUT2: max_act_lev=%d, num_thds=%d,
                    max_thds=%d\n",
                    omp_get_max_active_levels(),
                    omp_get_num_threads(), omp_get_max_threads());
        }
    }
    return 0;
}
```

На консоль выводит одна из нитей каждой вложенной области, помечая свой вывод номером нити, которая породила эту параллельную область. Вывод содержит текущие значения IS-переменных: количество возможных уровней вложенности параллельных областей (глобальное значение 4), количество нитей в текущей параллельной области (одна копия на область, в нашем случае значение равно 3), количество нитей, которые будут созданы в следующей параллельной области (одна копия на область, в примере в выводе OUT1 значение равно 4).

Вывод на консоль, помеченный OUT2, будет сделан одной нитью из внешней параллельной секции:

```
OUT2: max_act_lev=4, num_thds=2, max_thds=3
```

Из вывода видно, что в пределах внешней параллельной секции действительны свои копии двух последних переменных.

Начиная со спецификации стандарта OpenMP v. 3.0 возможно создание вложенных параллельных областей с переменным количеством нитей в каждой. Рассмотрите внимательно вывод на консоль примера 4.29 (для простоты вывод упорядочен):

```
thread (0,0)
thread (1,0)
thread (1,1)
thread (2,0)
thread (2,1)
thread (2,2)
```

Первое число соответствует номеру нити, породившей текущую (внешняя параллельная область состоит из трех нитей), второе число показывает номер нити во вложенной области. Мы видим, что нулевая нить породила параллельную область, выполняемую одной нитью, первая – двумя, вторая – тремя.

Следующий пример 4.30 демонстрирует поведение OpenMP-программы при включении и отключении возможностей динамического изменения количества нитей и вложения областей.

В примере реализованы четыре отдельные параллельные области и две вложенные. Скомпилировав этот код и выполнив его на двухядерном компьютере, мы получили такой результат:

```
Количество процессоров в системе = 2
Число нитей в динамической области = 2
Число нитей в нединамической области = 10
Динамика +
Вложение областей отключено, число нитей = 1
Вложение областей отключено, число нитей = 1
Вложение областей включено, число нитей = 2
Вложение областей включено, число нитей = 2
```


Динамика -

Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10
 Число нитей во вложенной области = 10

Пример 4.29. Создание вложенных параллельных областей с разным количеством нитей

```
#include <omp.h>
#include <stdio.h>
int main (void){
    int my_id;
    omp_set_dynamic(0);
    omp_set_nested(1);
    omp_set_max_active_levels(4);
    omp_set_num_threads(3);
    #pragma omp parallel private(my_id)
    {
        my_id = omp_get_thread_num();
        omp_set_num_threads(omp_get_thread_num()+1);
        #pragma omp parallel shared(my_id)
        {
            printf ("thread (%d, %d)\n", my_id,
                    omp_get_thread_num());
        }
    }
    return 0;
}
```

Пример 4.30. IS-переменные и функции времени выполнения

```
#include <stdio.h>
#include <omp.h>
int main(){
    printf("Количество процессоров в системе = %d\n",
           omp_get_num_procs());

    omp_set_dynamic(1);
    omp_set_num_threads(10);
    #pragma omp parallel // параллельная область 1
    {
        #pragma omp single
        printf("%d нитей в динамической области\n",
              omp_get_num_threads());
    }
}
```

Пример 4.30 (окончание). IS-переменные и функции времени выполнения

```
printf("\n");
omp_set_dynamic(0);
omp_set_num_threads(10);
#pragma omp parallel // параллельная область 2
{
    #pragma omp single
    printf("%d нитей в нединамической области \n",
        omp_get_num_threads());
}
printf("\n");
omp_set_dynamic(1); printf("Динамика +\n");
omp_set_nested(0);
omp_set_num_threads(10);
#pragma omp parallel // параллельная область 3
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Вложение областей отключено, %d нитей \n",
            omp_get_num_threads());
    }
}
printf("\n");
omp_set_nested(1);
#pragma omp parallel // параллельная область 4
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Вложение областей включено, %d нитей \n",
            omp_get_num_threads());
    }
}
printf("\n");
omp_set_dynamic(0); printf("Динамика -\n");
omp_set_nested(1);
omp_set_num_threads(10);
#pragma omp parallel // параллельная область 5
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("%d нитей во вложенной области\n",
            omp_get_num_threads());
    }
}
}
```

Для первой области включено динамическое создание потоков и установлено число потоков 10. По результатам работы программы видно, что при включенном динамическом создании потоков исполняющая среда OpenMP решила создать группу, включающую всего два потока, поскольку в распоряжении среды два процессора. Для второй параллельной области исполняющая среда OpenMP создала группу из 10 потоков, потому что динамическое создание потоков для этой области было отключено.

Результаты выполнения третьей и четвертой параллельных областей иллюстрируют следствия включения и отключения возможности вложения областей при включенной возможности динамического создания потоков. В третьей параллельной области вложение было отключено, поэтому для вложенной параллельной области не было создано никаких новых потоков — и внешняя, и вложенная параллельная области выполнялись двумя потоками. В четвертой параллельной области, где вложение было включено, для вложенной параллельной области была создана группа из двух потоков (т. е. в общей сложности эта область выполнялась четырьмя потоками). Процесс удвоения числа потоков для каждой вложенной параллельной области может продолжаться, пока не исчерпается пространство в стеке. На практике можно создать несколько сотен потоков, хотя связанные с этим издержки легко перевесят любые преимущества.

Для пятой параллельной области вложение параллельных областей разрешено, но динамическое создание потоков отключено. В этом случае для первой параллельной области создается группа из 10 потоков, затем при входе во вложенную параллельную область для каждого из этих 10 потоков создается группа также из 10 потоков. В общей сложности, 100 потоков выполняют вложенную параллельную область.

Обсудим назначение трех переменных среды выполнения, значения которых можно установить только до выполнения программы, для них не существует функций времени выполнения, изменяющих их значения.

Переменная среды `OMP_PROC_BIND` определяет привязку нити к вычислительному ядру. Если значение переменной равно `false`, то во время выполнения параллельной области запрещается перебрасывать нити с одного процессора (ядра) на другой. Если значение переменной равно `true`, то для балансировки нагрузки среда выполнения может принять решение о назначении нити другому процессору (ядру).

Переменная среды `OMP_STACKSIZE` определяет размер стека для порождаемых в параллельной области нитей. Переменная не определяет размер стека для основной нити. Размер можно задать в одном из следующих форматов:

```
size | sizeB | sizeK | sizeM | sizeG
```

Здесь `size` – это целое значение, а `B`, `K`, `M` или `G` означают единицы, в которых задан размер – байты, килобайты, мегабайты или гигабайты соответственно. По умолчанию размер стека задается в килобайтах. Значение переменной по умолчанию определяется реализацией.

Переменная среды `OMP_WAIT_POLICY` определяет, как ведет себя нить во время ожидания. Переменная принимает одно из следующих значений – `ACTIVE` | `PASSIVE`. Если значение переменной установлено в `ACTIVE`, то применяется механизм активного ожидания, т. е. нить не снимается с выполнения до истечения кванта времени, даже если она находится в режиме ожидания при синхронизации или по любому другому поводу. Если значение переменной равно `PASSIVE`, то нить может быть вытеснена с выполнения при ожидании. Значение переменной по умолчанию определяется реализацией.

4.8. Спецификации стандарта OpenMP v. 4.0

В июле 2013 года выпущен новый набор спецификаций OpenMP 4.0. Разработчики стандарта учли современные тенденции развития архитектур вычислительных систем в сторону поддержки дополнительных вычислителей с массовым параллелизмом (технология GPGPU использования графических процессоров для задач общего назначения, вычислительный сопроцессор Intel® Xeon Phi™ и т. д.). В OpenMP 4.0 введена поддержка ускорителей. Кроме того, в новый набор спецификаций внесены изменения в модели выполнения задач, добавлены механизм обработки ошибок и возможность определения пользовательских редукций в конце параллельной области.

Кратко опишем характерные черты новой спецификации.

- В архитектуре многих современных процессоров присутствуют блоки для выполнения параллельных по данным операций (SIMD-блоки), используемые для векторизации последовательных и параллелизованных циклов. API OpenMP 4.0 предоставляет механизмы для указания того, как распараллелить циклы с использованием инструкций SIMD. Кроме того, пользователь может определить функции, которые должны вызываться в блоках, использующих SIMD. Для поддержки этих возможностей определены директивы `omp simd`, `omp declare simd`, `omp for simd` и `pragma omp distribute simd`.

- Важной чертой новой спецификации является поддержка дополнительных аппаратных вычислительных устройств ряда производителей, например, GPU. API OpenMP 4.0 предоставляет механизмы, позволяющие указать область кода и/или данных, которые должны быть обработаны

с использованием другого вычислительного устройства, а также возможности отображения общей памяти хоста и памяти устройства. Для поддержки этих возможностей определены несколько директив `omp target`, директива `omp teams` и несколько директив распределения итераций цикла `omp distribute`. Кроме того, определены переменная среды (`OMP_DEFAULT_DEVICE`), несколько внутренних контрольных переменных и функции времени выполнения (`omp_set_default_device`, `omp_get_default_device`, `omp_get_num_devices`, `omp_get_num_teams`, `omp_get_team_num` и `omp_is_initial_device`).

- В новой спецификации появилась возможность специальной обработки исключений. API OpenMP 4.0 предоставляет средства для обработки ошибок (системных, времени выполнения и ошибочных ситуаций, определенных пользователем). Пользователю предоставлена возможность определять точки отмены параллельного выполнения, а также условное прекращение параллельного выполнения. Для поддержки этих возможностей определены директивы `omp cancel` и `omp cancellation point`. Кроме того, определены переменная среды (`OMP_CANCELLATION`), несколько внутренних контрольных переменных и функция времени выполнения `omp_get_cancellation`.

- В новой спецификации появилось понятие привязки потоков. API OpenMP 4.0 предоставляет механизмы для указания того, где именно выполнять потоки OpenMP. Для поддержки этих возможностей определены опция `proc_bind` директивы `omp parallel`, переменная среды (`OMP_PLACES`), несколько внутренних контрольных переменных и функция времени выполнения `omp_get_proc_bind`.

- Несколько изменений новой спецификации относятся к программированию параллелизма на уровне задач. API OpenMP 4.0 предоставляет возможность группировки задач (директива `omp taskgroup`) для обеспечения их гибкой синхронизации. Группа задач может быть прервана целиком, что полезно, например, при поиске для остановки совместно выполняемого набора задач-ищек. API OpenMP 4.0 предоставляет гибкий механизм синхронизации задачи с задачей путем описания зависимостей (опция `depend` директивы `omp task`). API OpenMP 4.0 позволяет определять точки перепланирования для задач, которые не привязаны к нитям.

- Кроме поддержки редукции в конце параллельной области на основе базовых операций языка в API OpenMP 4.0 добавлены механизмы для определения редукций пользователем. Для поддержки этих возможностей добавлена директива `omp declare reduction`, которая позволяет ввести именованное правило редукции. Это правило может затем использоваться в опции `reduction` директивы `omp parallel`.

- В API OpenMP 4.0 расширены возможности директивы `omp atomic`. С помощью новой опции `seq_cst` этой директивы принудительно применяется более сильная модель последовательного согласования общей памяти, поддерживающей атомарный доступ.
- В Си/Си++ API OpenMP 4.0 для распределения памяти введена поддержка понятия секции массива (array section).
- В API OpenMP 4.0 введена новая переменная среды `OMP_DISPLAY_ENV`, помогающая отобразить значения всех переменных среды.
- В API OpenMP 4.0 реализована поддержка стандарта Fortran 2003.

4.9. Оптимизация программ OpenMP

Распараллеливая программу, мы ожидаем уменьшения времени ее исполнения в многопроцессорной (многоядерной) среде. Теоретические оценки потенциального ускорения обсуждаются в гл. 5. Здесь сделаем только несколько важных замечаний, непосредственно связанных с технологией OpenMP.

Первая проблема заключается в оптимизаторских возможностях компилятора. Современные компиляторы обладают сложными алгоритмами анализа программ с целью оптимизации кода. Практика показывает, что, например, компиляция вычислительной программы с помощью компилятора фирмы Intel с ключом `-O2` может на 10–15 % уменьшить время выполнения по сравнению с временем выполнения этой же программы, откомпилированной с этим же ключом компилятором `gcc`. Более того, правильный подбор ключей компиляции также способен существенно уменьшить время выполнения программы. Однако подключение библиотеки OpenMP заведомо усложняет анализ программы и приводит к снижению оптимизационных возможностей компиляторов. Конечно, на учебных примерах обнаружить снижение уровня оптимизации сложно, но при компиляции реальных вычислительных программ с ключом `-openmp` и без него можно заметить разницу во времени выполнения обеих версий не в пользу OpenMP.

Вторая проблема связана с расходами на поддержку многопоточности во время выполнения OpenMP-программы. Дополнительное время необходимо на создание нити в начале и уничтожение ее в конце каждой параллельной секции, на создание и/или инициализацию копий локальных переменных, распределение работы между нитями. Поэтому даже если алгоритм не подразумевает никаких затрат на синхронизацию внутри нитей и вызов функций времени выполнения, параллельная программа, откомпи-

лированная для OpenMP и отработавшая с одной нитью в каждой параллельной секции, затратит на свое выполнение времени больше, чем соответствующая последовательная программа.

Третья проблема связана с задержками при синхронизации между нитями, а также с синхронизацией общих данных между временным представлением нити и общей памятью. Сюда же следует отнести вызовы функций времени выполнения.

Все перечисленные проблемы увеличивают время выполнения OpenMP-программы и могут свести на нет предполагаемые выгоды от параллельного выполнения вычислений.

В табл. 4.4 указано ориентировочное время выполнения различных операций в тактах процессора.

Таблица 4.4

Операция	Минимальные затраты, такты	Масштабируемость (зависимость от количества нитей)
Попадание в L1-кэш	1–10	Постоянная
Вызов функции	10–20	Постоянная
Запрос номера потока	10–50	Постоянная, линейная
Целочисленное деление	50–100	Постоянная
Статическое распределение работы в цикле for (без синхронизации в теле цикла)	100–200	Постоянная
Непопадание в кэш	100–300	Постоянная
Синхронизация	200–500	Линейная, логарифмическая
Создание параллельной области (omp parallel)	500–1000	Линейная

При проектировании OpenMP-программы желательно соблюдать следующие простые рекомендации:

- не создавайте параллельные секции без необходимости, нет смысла распараллеливать сложение десяти чисел;
- избегайте лишних точек синхронизации, лучше несколько изменить логику алгоритма;
- используйте `nowait` везде, где это возможно по алгоритму;
- минимизируйте количество общих данных.

4.10. Ограничения OpenMP

Как мы убедились, технология OpenMP описывает параллельные потоки на более высоком уровне абстракции, чем API ОС или библиотеки

стандарта POSIX. В OpenMP нельзя произвольно создавать нити и распределять данные во время исполнения программы. В то же время OpenMP обладает широкими встроенными возможностями декомпозиции данных в циклах, назначения задач отдельным нитям, синхронизации и управления общими данными.

В OpenMP-программе, как в любой многопоточной программе, возможно возникновение состояния «гонок» и блокировок, не выявляемых при компиляции. Компилятор OpenMP не анализирует логическую правильность кода, корректность синхронизаций и использования общей памяти. Выделение проблемных участков алгоритма и разработка кода с защищенными корректно взаимодействующими нитями – обязанность программистов.

Существует ряд ограничений на алгоритмы при применении технологии OpenMP:

1. При использовании OpenMP невозможна синхронизация нитей в парадигмах «потребитель – производитель», «клиент – сервер» и большинства задач условной синхронизации.

2. В OpenMP нет объектов, подобных семафорам и событиям, следовательно, невозможно сложное планирование выполнения нитей средствами OpenMP.

3. Нельзя программировать с помощью OpenMP задачи, использующие изменение приоритета исполнения потока.

4. При выполнении OpenMP-программы в фоновом режиме производится большое количество работы, связанное с созданием/уничтожением нитей, их синхронизацией, распределением нагрузки и т. д. Пользователь получает об этом мало информации и не может эффективно влиять на происходящее.

Узким местом всех многопоточных технологий является отладка программ. Существует ряд инструментов корпорации Intel, таких как Intel Thread Checker, которые обеспечивают возможность анализа работы многопоточного приложения, использующего OpenMP. Данный отладчик встраивает свои функции в код программы и при ее запуске анализирует состояние потоков, переключателей и доступ к памяти. Отладчик способен выявить широкий спектр ошибок, таких как:

- неверный доступ к памяти;
- жесткие и мягкие блокировки;
- состояние «гонок»;
- ошибки синхронизации.

Подробно работа с отладчиком описана, например, в [28].

Однако, несмотря на перечисленные ограничения, следует подчеркнуть, что реализации OpenMP зарекомендовали себя как очень надежные. OpenMP – простая, но мощная технология распараллеливания приложений.

Она позволяет реализовать эффективное параллельное выполнение как циклов, так и функциональных блоков кода. OpenMP легко интегрируется в существующие приложения, включается/выключается одним параметром компилятора, идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими циклами. OpenMP позволяет эффективно использовать вычислительную мощь многоядерных процессоров.

Контрольные вопросы и задания

1. Для каких целей была разработана технология программирования OpenMP? Опишите основную концепцию написания программ с использованием OpenMP.
2. Опишите модель памяти OpenMP-программы.
3. Опишите модель выполнения OpenMP-программы. Какие составляющие используются для управления средой выполнения?
4. Каким образом и для чего используются директивы OpenMP?
5. Опишите формат директивы OpenMP `#pragma omp parallel`.
6. Перечислите директивы распределения работы по нитям параллельной секции. Опишите основное назначение каждой из них. Ответ проиллюстрируйте примерами.
8. Опишите формат директивы OpenMP `#pragma omp for`. Перечислите опции этой директивы. Поясните, какие циклы можно распараллеливать, а какие нельзя. Почему? Ответ проиллюстрируйте примерами.
9. Какие директивы синхронизации используются в OpenMP?
10. Расскажите о модели согласования памяти, принятой в OpenMP. Каковы особенности использования директивы `#pragma omp flush`?
11. Что такое переменные среды OpenMP и внутренние контрольные переменные? Перечислите основные функции времени выполнения.
12. Какие алгоритмы планирования возможны в OpenMP?
12. Какие методы используются для оптимизации программ, написанных с использованием OpenMP?
13. Какие ограничения системы программирования OpenMP вы можете перечислить?

Задачи

Для приведенных ниже задач следует соблюдать следующий порядок работы:

1. Разработать и реализовать алгоритм решения задачи с помощью последовательной программы и протестировать его на нескольких примерах.

2. Разработать алгоритм решения задачи с учетом разделения вычислений между несколькими потоками. Избегать ситуаций изменения одних и тех же общих данных несколькими потоками. Составить схему потоков.

3. Реализовать алгоритм в среде OpenMP и протестировать его на нескольких примерах. Количество потоков является входным параметром, при этом размерность задачи может быть не кратна количеству потоков.

4. Сравнить объемы программного кода, сложность программирования отладки, производительность последовательной и многопоточной версии.

4.1. Дана последовательность символов $C = \{c_0, \dots, c_{n-1}\}$. Создать OpenMP-приложение, определяющее, является ли строка полиндромом (полиндром – фраза, читающаяся справа налево и слева направо одинаково, без учета пробелов). Количество символов и потоков является входным параметром программы.

4.2. Даны результаты сдачи экзамена по курсу «Средства разработки параллельных программ» по студенческим группам. Требуется создать OpenMP-приложение, вычисляющее средний балл на курсе и средний балл каждой группы. Количество потоков и количество групп являются входными параметрами программы.

4.3. Охранное агентство разработало новую систему управления электронными замками. Для открытия двери клиент обязан произнести произвольную фразу из $N \gg 1$ слов. В этой фразе должно встречаться заранее оговоренное слово, причем только один раз. Требуется создать OpenMP-приложение, управляющее замком.

4.4. Дан список студентов по группам. Требуется создать OpenMP-приложение для определения количества студентов с фамилией Иванов и студентов с фамилией Петров. В какой группе разница между количеством Ивановых и количеством Петровых максимальна? Количество потоков и количество групп являются входными параметрами программы.

4.5. Среди студентов СФУ проведен опрос с целью определения процента студентов, знающих точную формулировку правила Буравчика. В результате собраны данные о количестве знатоков в каждом институте по группам. Известно, что всего в СФУ обучается 33 000 студентов. Требуется создать OpenMP-приложение для определения процента знающих правило Буравчика студентов в каждом институте и во всем университете. Количество потоков, групп и институтов является входным параметром программы.

4.6. Дана последовательность натуральных чисел $C = \{c_0, \dots, c_{n-1}\}$. Создать OpenMP-приложение для вычисления общей суммы и всех промежуточных сумм.

4.7. Найти определитель невырожденной матрицы A .

4.8. Найти алгебраическое дополнение для каждого элемента невырожденной матрицы A .

4.9. Найти обратную матрицу для невырожденной матрицы A .

4.10. Найти LU -разложение невырожденной матрицы A , где L – нижнетреугольная матрица с единицами на главной диагонали, U – верхнетреугольная матрица.

4.11. Решить СЛАУ $Ax=f$ методом Гаусса.

4.12. Решить СЛАУ $Ax=f$ методом простой итерации.

4.13. Решить СЛАУ $Ax=f$ методом Зейделя.

4.14. Решить СЛАУ $Ax=f$ методом сопряженных градиентов.

4.15. Дана неупорядоченная последовательность чисел $C = \{c_0, \dots, c_{n-1}\}$.

Отсортировать C :

а) методом обменной сортировки со слиянием Бэтчера;

б) методом быстрой сортировки с использованием регулярного набора образцов;

с) методом пузырьковой сортировки;

д) методом сортировки Шелла.

4.16. *Задача о сглаживании изображения.* Дано черно-белое изображение в виде матрицы чисел-пикселей $m \times n$. Каждый пиксель имеет значение 1 (освещен) или 0 (не освещен). Изображение требуется сгладить, убрав «пики» и «зазубрины». Для этого применяется итеративный алгоритм. Начав с исходного изображения, затемняют все пиксели, у которых как минимум d соседей не освещены. Каждый пиксель рассматривается независимо. Затем эта процедура продлевается с полученным изображением и повторяется до тех пор, пока изображение не перестанет меняться. Числа m , n и d являются параметрами задачи. Исходное изображение хранится в одном файле.

4.17. *Задача о классификации.* Дано множество объектов X , разделенных некоторым образом на классы. (Объект можно представлять точкой в n -мерном пространстве, координаты точки – численные характеристики свойств объекта, по которым проводится классификация.)

Пусть Y – множество номеров (или наименований) классов. Существует неизвестная целевая зависимость – отображение $f: X \rightarrow Y$, значения которого известны только на объектах конечной обучающей выборки X^m .

Требуется построить алгоритм $A: X \rightarrow Y$, способный классифицировать произвольный объект из X .

Решить задачу кластеризации с помощью алгоритма максимального расстояния.

Проверку осуществить на известных наборах данных для классификации, например, на ирисах Фишера. Большой выбор данных представлен на сайте <http://archive.ics.uci.edu/ml/datasets.html>

4.18. *Метод Якоби для уравнения Пуассона.* Написать программу, решающую задачу Дирихле для уравнения Пуассона на квадрате $\Omega = [0,1] \times [0,1]$

$$\Delta u = f(x, y), \quad (x, y) \in \Omega,$$

$$u(x, y) = u_0, \quad (x, y) \in \Gamma$$

методом Якоби:

$$u_{i,j}^{(k+1)} = \frac{1}{4} \left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - h^2 f_{i,j} \right), \quad i, j = 1, \dots, n-1, \quad k = 1, 2, \dots$$

$$u_{i,j}^{(k)}|_{\Gamma} = u_0(x_i, y_j), \quad \forall k$$

на равномерной сетке $\omega^h = \{(x_i, y_j) : x_i = ih, y_j = jh, i, j = 0, \dots, n\}$.

4.19. *Метод Гаусса – Зейделя для уравнения Пуассона.* Написать программу, решающую задачу Дирихле для уравнения Пуассона из задачи 4.18 методом Гаусса – Зейделя:

$$u_{i,j}^{(k+1)} = \frac{1}{4} \left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)} - h^2 f_{i,j} \right), \quad i, j = 1, \dots, n-1, \quad k = 1, 2, \dots$$

$$u_{i,j}^{(k)}|_{\Gamma} = u_0(x_i, y_j), \quad \forall k.$$

4.20. *Метод Рундсона для уравнения теплопроводности.* Написать программу, решающую уравнение теплопроводности в области $\Omega = [0, T] \times [0, L]$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad (t, x) \in (0, T] \times (0, L),$$

$$u(0, x) = \mu(x), \quad x \in [0, L],$$

$$u(t, 0) = \mu_1(t), \quad u(t, L) = \mu_2(t), \quad t \in [0, T]$$

на равномерной сетке методом Рундсона:

$$\frac{1}{2\tau} (u_i^{k+1} - u_i^k) = \frac{1}{h} (u_{i-1}^k - 2u_i^k + u_{i+1}^k),$$

с условием устойчивости $\tau < h^2$, где h – шаг сетки по пространству; τ – шаг по времени; $\mu(t)$, $\mu_1(t)$, $\mu_2(t)$ – заданные функции.

4.21. *Метод Якоби для уравнения теплопроводности.* Написать программу, решающую уравнение теплопроводности в области $\Omega = [0, T] \times [0, L]$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(t, x), \quad (t, x) \in (0, T] \times (0, L),$$

$$u(0, x) = \mu(x), \quad x \in [0, L],$$

$$u(t, 0) = \mu_1(t), \quad u(t, L) = \mu_2(t), \quad t \in [0, T]$$

методом Якоби:

$$\frac{1}{\tau}(u_i^{k+1} - u_i^k) = \frac{1}{h}(u_{i-1}^k - 2u_i^k + u_{i+1}^k) + F_i^k$$

с условием устойчивости $\tau < 2h^2$, где h – шаг сетки по пространству, τ – шаг по времени, $\mu(t)$, $\mu_1(t)$, $\mu_2(t)$ – заданные функции.

4.22. *Метод Годунова.* Написать программу, решающую систему уравнений, записанную в скоростях v и напряжениях σ :

$$\frac{\partial v}{\partial t} = \frac{\partial \sigma}{\partial x}, \quad (t, x) \in (0, T] \times (0, L),$$

$$\frac{\partial \sigma}{\partial t} = \frac{\partial v}{\partial x}, \quad (t, x) \in (0, T] \times (0, L),$$

$$v(0, x) = \sigma(0, x) = 0, \quad x \in [0, L],$$

$$\sigma(t, 0) = \sigma_0(t), \quad v(t, L) = 0, \quad t \in [0, T],$$

где $\sigma_0(t)$ – заданная функция.

Для решения задачи воспользуйтесь методом Годунова типа «предиктор – корректор» на равномерной сетке.

На шаге «предиктор» вычисляются промежуточные значения $S_{i+1/2}$ и $V_{i+1/2}$ по следующим формулам:

$$S_{i+1/2} = (\sigma_{i+1} + \sigma_i + v_{i+1} - v_i) / 2,$$

$$V_{i+1/2} = (v_{i+1} + v_i + \sigma_{i+1} - \sigma_i) / 2.$$

На шаге «корректор» вычисляются скорости и напряжения (крышка над символом означает следующий шаг по времени):

$$\hat{v}_i = v_i + \frac{\tau}{h}(S_{i+1/2} - S_{i-1/2}),$$

$$\hat{\sigma}_i = \sigma_i + \frac{\tau}{h}(V_{i+1/2} - V_{i-1/2}).$$

Для граничных условий используются соотношения:

$$S_{1/2} = \sigma_0(t), \quad V_{1/2} = -S_{1/2} + v_1 + \sigma_1,$$

$$V_{L+1/2} = 0, \quad S_{L+1/2} = V_{L+1/2} + \sigma_L - v_L.$$

Условие устойчивости метода: $\tau < h$, где h – шаг сетки по пространству; τ – шаг по времени.

4.23. *Метод «крест» Неймана – Рихтмайера.* Написать программу, решающую систему уравнений из задачи 4.22 методом «крест» Неймана – Рихтмайера:

$$v_{i+1/2}^{k+1/2} = v_{i+1/2}^{k-1/2} + \frac{\tau}{h}(\sigma_{i+1}^k - \sigma_i^k),$$

$$\sigma_{i+}^{k+1} = \sigma_i^k + \frac{\tau}{h}(v_{i+1/2}^{k-1/2} - v_{i-1/2}^{k+1/2}).$$

Условие устойчивости $\tau < h$, где h – шаг сетки по пространству; τ – шаг по времени.

Глава 5 | СОГЛАСОВАННОЕ ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

В главе рассмотрены проблемы создания приложений для ВС с распределенной памятью на основе механизма передачи сообщений в классе согласованного параллельного программирования. Кратко обсуждаются вопросы, связанные с исследованием информационных зависимостей и оценками внутреннего параллелизма алгоритмов, даются понятия ускорения, эффективности, масштабируемости. Описаны двухточечные и коллективные взаимодействия процессов, подходы к оценке времени, необходимого на передачу сообщений в кластерных системах. В конце главы рассмотрены основные вычислительные задачи линейной алгебры и схемы возможных параллельных алгоритмов для их решения.

5.1. Проблемы программирования для вычислительных систем с распределенной памятью

Если ресурс разделяемый, то встает проблема взаимного исключения и условной синхронизации процессов для использования этого ресурса. Напротив, при программировании в системах с распределенной памятью каждая переменная является локальной для какого-либо одного процесса. Проблема взаимного исключения становится своей противоположностью – проблемой обмена значениями. Реализация механизма условной синхронизации также принципиально меняется.

Поскольку при распределенной архитектуре на пути к результатам вычислений другого процессора лежит связующая сеть, то необходимо определить интерфейсы с сетью связи. Как правило, выделяют специальные сетевые операции, включающие синхронизацию взаимодействующих процессов, – *примитивы передачи сообщений*. Передачу сообщений можно рассматривать как обобщение семафоров для перенаправления данных и обеспечения синхронизации.

При передаче сообщений процессы разделяют не переменные, а каналы. *Канал* – абстракция сети связи, обеспечивающая физический путь между процессами. Реализация канала принципиально разная, если он соединяет различные процессы на одной машине (реализация с помощью разделяемых переменных) и если общение между процессами осуществляется с помощью связующей сети. В последнем случае каналы – единственные объекты, которые разделяют процессы.

В рамках распределенного программирования варьируются следующие характеристики взаимодействия процессов:

- направленность потока информации по каналу (одно- или двунаправленный);
- тип взаимодействия (синхронное/асинхронное).

В результате рассматриваются четыре механизма взаимодействия распределенных процессов:

1) асинхронная передача сообщений: однонаправленный канал, асинхронное взаимодействие;

2) синхронная передача сообщений: однонаправленный канал, синхронное взаимодействие;

3) удаленный вызов процедур (Remote procedure call, RPC): двунаправленный канал, асинхронное взаимодействие;

4) рандеву: двунаправленный канал, синхронное взаимодействие.

Эти механизмы взаимозаменяемы, но парадигмы «взаимодействующие равные» и «производитель – потребитель» удобнее реализовать с использованием передачи сообщений, а «клиент – сервер» и «управляющий – рабочий» – с помощью RPC и рандеву.

Передача сообщений

При передаче сообщений канал представляется очередью FIFO, реализующей однонаправленную передачу. Доступ к каналу производится с использованием двух примитивов – послать и получить сообщение.

При асинхронной передаче сообщений каналы подобны семафорам, переносящим данные. В такой трактовке примитив отправки сообщения `send` соответствует операции `v()` (сигнализирует, что событие – запись в канал сообщения – произошло), а примитив получения сообщения `receive` – операции `P()` (дождаться, пока не произойдет событие – в канале не появится сообщение – и прочесть его). При этом число сообщений в очереди соответствует значению семафора.

Таким образом, при асинхронной передаче сообщений канал можно представить как виртуальную очередь сообщений, которые уже отправлены, но еще не получены. Рассмотрим нотацию для объявления канала:

```
chan ch(type_1, ..., type_n);
```

Здесь `ch` – имя канала; `type_i` – типы полей данных в передаваемых по каналу сообщениях. Допустимо использовать массивы каналов, например, массив каналов, нумерованных от 0 до $n-1$, объявляется следующей строкой:

```
chan result [n] (int);
```

Процесс отправляет сообщение каналу `ch`, выполняя операцию:

```
send ch(expr_1, ..., expr_n);
```

Необходимо соответствие типов выражений `expr_i` и типов в декларации канала. При выполнении операции `send` сначала вычисляются `expr_i`, затем полученные выражения присоединяются к концу очереди, связанной с каналом `ch`. Теоретически эта очередь не ограничена, поэтому выполнение процесса после этого продолжается сразу. Таким образом, в объявленной нотации операция `send()` является *неблокирующей* операцией. На практике ресурсы системы всегда не бесконечны, поэтому существует достаточное количество реализаций канала, учитывающих его ограниченность.

Процесс получает сообщение из канала `ch`, выполняя операцию

```
receive ch(var_1, ..., var_n);
```

Переменные `var_i` должны иметь тот же тип, что и соответствующие поля декларации канала. При выполнении операции `receive()` принимающий процесс приостанавливается до тех пор, пока в очереди канала не появится хотя бы одно сообщение. Затем из очереди удаляется первое сообщение, а значения его полей присваиваются переменным `var_i`. Операция `receive()` всегда является блокирующей операцией. При этом процесс, выполняющий операцию `receive()`, не обязан применять активное ожидание для опроса канала.

Предполагается, что доступ к содержимому канала является неделимым, а сообщения передаются без ошибок. Таким образом, каждое переданное в канал сообщение будет, в конце концов, принято, причем без искажений.

Каналы используются процессами совместно, поэтому объявляются как глобальные переменные.

Иногда получающий процесс можно занять полезной работой, пока он дожидается появления в канале сообщения, поэтому над каналом разумно определить функцию

```
empty(ch);
```

которая возвращает значение `true`, если канал пуст, и `false` в противном случае. Отметим существенное отличие операции `empty(ch)` от подобной операции с очередью для условной переменной монитора. Процесс может выполнить проверку очереди, выяснить, что очередь не пуста, но к моменту продолжения работы окажется, что другой процесс уже опустошил очередь, и наоборот, значение `true` не гарантирует, что в момент выполнения процессом следующего действия очередь будет по-прежнему пуста.

При синхронной передаче сообщений отправка сообщений приводит к блокировке процесса-отправителя до тех пор, пока сообщение не будет получено. Определим для синхронной передачи сообщений примитив

```
synch_send(expr_1, ..., expr_n);
```

с аргументами, аналогичными `send()`. Преимущество такой передачи состоит в том, что размер канала связи ограничен, при этом процесс-отправитель может поставить в очередь любого канала не более одного сообщения. Пока это сообщение не получено, процесс не может продолжить работу. При такой организации каналом является просто очередь адресов сообщений, ожидающих отправки.

Синхронная передача обладает двумя недостатками. Во-первых, при синхронной передаче снижается параллельность выполнения программ. Сообщения не накапливаются в очереди, а происходит задержка процесса-отправителя до той поры, пока процесс-получатель не выполнит прием. Во-вторых, синхронная организация обмена подвержена взаимным блокировкам. Выполнение кода из примера 5.1 гарантированно закончится мертвой блокировкой на операторах `synch_send()`. Однако при замене операторов `synch_send()` примитивами асинхронной передачи `send()` программы будут работать успешно.

Пример 5.1. Мертвая блокировка

```
channel in1(int), in2(int);
process P1 {
  int val_1=1, val_2;
  synch_send in2(val_1);
  receive in1(val_2);
}
```

```
process P2 {
  int val_1, val_2=2;
  synch_send in1(val_2);
  receive in2(val_1);
}
```

Если любой процесс может отправить данные в любой канал и принять их из любого канала, то такой канал называется *почтовым ящиком*. Если у канала много отправителей, но один получатель, то канал называют *входным портом*. Если у канала один отправитель и один получатель, то такой канал называют *каналом связи*.

Низкоуровневая реализация каналов и примитивов приема/передачи сообщений выходит за рамки нашего обсуждения. Кратко поясним лишь следующее. Наиболее распространены два метода коммуникации.

1. *Передача сообщений как неделимых (атомарных) блоков* информации. Процесс-отправитель готовит данные, определяет адресата, пополняя сообщение служебной информацией, и запускает операцию пересылки. Процесс-адресат полностью принимает пакет и только затем, анализируя служебную информацию, решает его дальнейшую судьбу – пересылать далее по маршруту или использовать по назначению.

2. *Пересылка пакетов* предполагает разбиение сообщения на блоки информации меньшего размера и сопровождение пакета служебной информацией. Принимающий процесс может пересылать пакет далее по

маршруту сразу после его получения. Сообщение собирается и восстанавливается только в конечном пункте. Такой метод, как правило, быстрее.

Отметим также, что конкретные языковые конструкции очень гибки, например, библиотека MPI насчитывает восемь основных функций только двухточечной отправки сообщений, а всего с передачей «точка – точка» связано более сорока функций.

Удаленные операции

Несмотря на то, что с помощью передачи сообщений можно программировать парадигму «клиент – сервер», такая реализация будет не очень удачна. Например, двусторонний поток данных между клиентом и сервером приходится программировать двумя явными передачами сообщений по двум отдельным каналам, кроме того, часто требуется еще отдельный канал подтверждения получения ответа. Все это приводит к увеличению числа каналов.

Удаленные операции сочетают в себе основные черты мониторов и синхронной передачи сообщений. Операции используют двунаправленный канал от вызывающего процесса к процессу, обслуживающему вызов.

Предполагается, что процессы могут «экспортировать» операции. В случае *RPC* вызывающий процесс примитивом *call* инициализирует запуск нового процесса с требованием исполнить его экспортируемую процедуру. При использовании механизма *рандеву* вызывающий процесс инициализирует встречу с существующим процессом примитивом ввода *in*, который ожидает вызова, обрабатывает его и возвращает результат¹.

Рандеву и *RPC* обычно применяют для программирования распределенных приложений. Проблемы создания таких приложений выходят за рамки нашего изложения и далее не обсуждаются.

5.2. Оценка эффективности параллельных алгоритмов

Как отмечалось в гл. 1, согласованное параллельное программирование предполагает построение параллельных приложений, способных совместно быстро решить поставленную задачу. Большое место в этом классе занимают сложные научно-технические, оптимизационные, комбинаторные задачи.

При разработке параллельных приложений для их решения встает вопрос об оценке эффективности распараллеливания. Обычно рассматриваются два основных подхода:

¹ Сравните работу сервера баз данных и использование DHTML с обработкой экранных форм.

- 1) оценка внутреннего параллелизма выбранного алгоритма;
- 2) получение оценок максимально возможного ускорения процесса решения конкретной задачи.

В данном параграфе мы кратко обсудим эти темы, их подробное изложение можно найти, например, в монографиях [9–14, 35].

Время выполнения программы на одном вычислителе¹ T_1 в общем случае состоит из времени $T_1^{calc}(N)$, которое затрачивается собственно на вычисления, и времени обращения к памяти $T_1^{mem}(N)$:

$$T_1(N) = T_1^{calc}(N) + T_1^{mem}(N). \quad (5.1)$$

Здесь N характеризует вычислительную сложность (размерность) задачи.

Время выполнения параллельной программы на p вычислителях состоит из времени $T_p^{calc}(N)$, затрачиваемого на выполнение вычислений, времени работы с памятью $T_p^{mem}(N)$ и времени $T_p^{over}(p, N)$, затрачиваемого на дополнительные издержки, связанные с параллельным выполнением алгоритма:

$$T_p(N) = T_p^{calc}(N) + T_p^{mem}(N) + T_p^{over}(p, N). \quad (5.2)$$

Внутренний параллелизм алгоритма

Для текущего обсуждения предположим некоторую идеальную ситуацию параллельного решения задачи. Во-первых, будем считать, что в нашем распоряжении столько вычислителей, сколько необходимо для решения задачи. Во-вторых, не будем учитывать время $T_p^{mem}(N)$, необходимое для считывания из памяти и записи в память (в том числе параллельно). В-третьих, будем пренебрегать и временем $T_p^{over}(p, N)$, необходимым на операции обмена данными или синхронизацию. Таким образом, мы будем обсуждать только время $T^{calc}(N)$, затрачиваемое собственно на вычисления.

Решение *вычислительной* задачи находится путем выполнения множества простых операций, имеющих небольшое число аргументов (обычно предполагается, что оно не больше двух). Представим множество операций, выполняемых в исследуемом алгоритме решения, и существующие между операциями информационные зависимости в виде *ациклического ориентированного графа* $G=(V, R)$. В качестве множества вершин V графа G

¹ Под вычислителем здесь и далее понимается устройство, которое непосредственно выполняет параллельную программу. Это может быть процессор в многопроцессорной системе или вычислительное ядро в многоядерной архитектуре.

рассмотрим множество всех выполняемых операций алгоритма. Из вершины $v_i \in V$ в вершину $v_j \in V$ проводится дуга $r_{ij} \in R$, если операция j использует результат операции i . Если соответствующие операции могут выполняться независимо, то дуга не проводится.

Для простоты изображения графа без потери общности будем считать, что в случае, когда аргументами операции являются начальные данные, соответствующие входные дуги отсутствуют. Альтернативой этому может быть введение для входных данных специальных *вершин ввода*, что для большого количества операций и малого количества входных параметров сильно усложняет граф. Если результат операции нигде не используется, то выходные дуги отсутствуют. Если вершина не имеет ни одной входящей дуги, то она называется *входной*, если вершина не имеет ни одной выходящей дуги, то она называется *выходной*.

Обозначим через $d(G)$ диаметр графа (длину его наименьшего пути). Для решения конкретной задачи в общем случае может существовать несколько графов, отражающих информационные зависимости различных алгоритмов. Операции алгоритма, между которыми нет пути в рамках выбранной схемы вычислений, могут быть выполнены *параллельно*.

На рис. 5.1 для примера приведен граф, отражающий информационные зависимости одной из схем вычисления следующего выражения:

$$f(x, y) = x^3 + y^3 + x^2y + xy^2 + x^2 + y^2 + xy + x + y + 1.$$

Для вычисления $f(x, y)$ необходимо выполнить 16 операций – 9 сложений и 7 умножений. *Входная степень вершин* графа (количество входящих дуг) не превышает двух, в то время как максимальная *выходная степень* (количество выходящих дуг) вершины, соответствующей операции xy , равна трем. Диаметр графа равен четырем $d(G) = 4$.

Даже визуальный анализ графа показывает, что можно выделить пять таких групп вершин, для которых все операции в группе можно выполнять параллельно. Группу принято называть *ярусом*, количество вершин в ярусе – *шириной яруса*. При наличии достаточного количества вычислителей алгоритм можно выполнить «по ярусно», т. е. последовательно продвигаясь от входного яруса, одновременно выполнять все операции каждого яруса. Для оценки времени выполнения *вычислительной части* алгоритма предполагается, что любая операция выполняется за единицу времени. Тогда выполнение алгоритма по ярусам по времени будет оптимальным.

Оценка T_1^{calc} определяет время выполнения вычислений алгоритма при использовании одного вычислителя и представляет тем самым время выполнения вычислений последовательного варианта алгоритма решения

задачи. Ясно, что $T_1^{calc}(G) = |V|$, где V – количество вершин вычислительной схемы G .

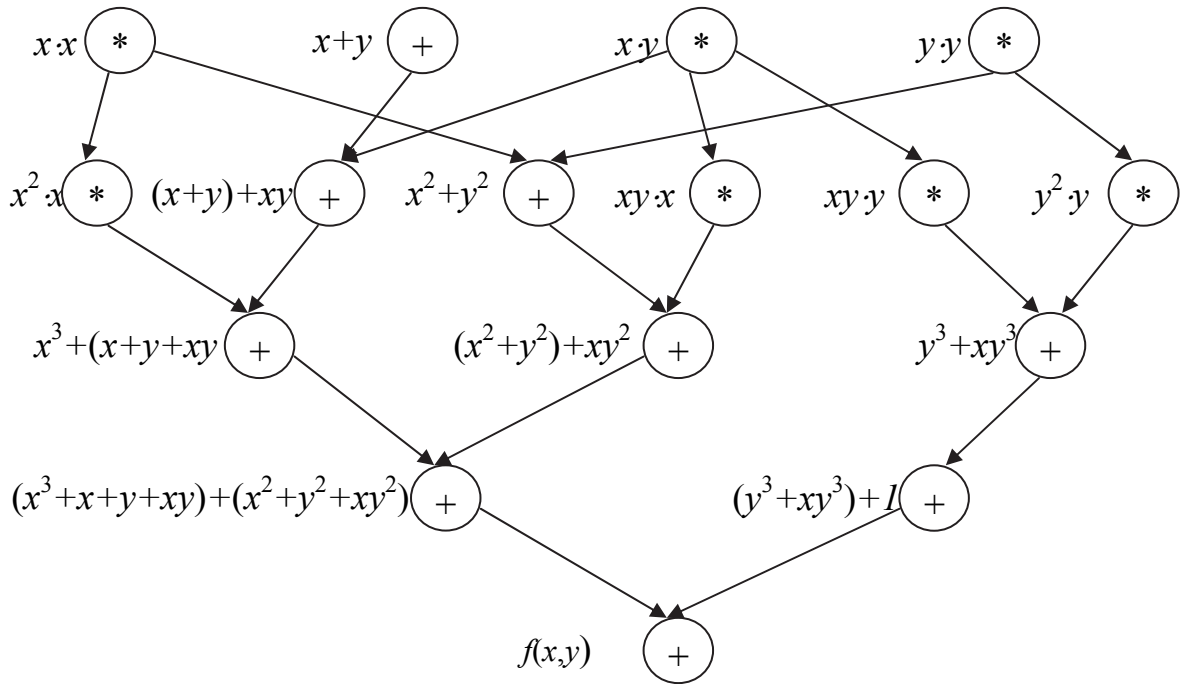


Рис. 5.1. Информационные зависимости одной из вычислительных моделей $f(x,y)$

На практике при оценке T_1^{calc} обычно ограничиваются рассмотрением только одного выбранного алгоритма решения задачи. Однако для оценки эффективности параллельного решения время последовательного решения желательно определять с учетом *различных последовательных алгоритмов*, т. е. $T_1^{*calc} = \min_{\forall G \in \Gamma} T_1^{calc}(G)$, где операция минимума берется по множеству всех возможных последовательных алгоритмов решения данной задачи. Оценка T_∞^{calc} определяет минимально возможное время выполнения параллельного алгоритма при использовании неограниченного количества вычислителей. Наконец, зная доступное количество вычислителей p , можно получить оценку T_p^{calc} .

Приведем без доказательства теоретические положения, характеризующие свойства оценок времени выполнения параллельного алгоритма [9, 14, 44].

Теорема 1. Минимально возможное время выполнения параллельного алгоритма определяется длиной максимального пути вычислительной схемы алгоритма $T_\infty^{calc}(G) = d(G) + 1$.

Теорема 2. Пусть для некоторой вершины вывода в вычислительной схеме алгоритма существует путь из каждой входной вершины. Кроме того, пусть входная степень вершин схемы не превышает 2. Тогда минимально возможное время выполнения параллельного алгоритма ограничено снизу значением $T_{\infty}^{calc}(G) \geq T_{\infty}^{*calc}(G) = \log_2 n$, где n – количество вершин входа (т. е. количество входных данных).

Заметим, что в случае вычислительной схемы на рис. 5.1 оценка снизу точно не достигается: $T_{\infty}^{calc}(G_{(f)}) = 5 > \log_2 2 = T_{\infty}^{*calc}(G_{(f)})$. Это связано с тем, что в нашем примере для большинства вершин первого яруса выходная степень больше единицы, т. е. одна операция «порождает» несколько.

Теорема 3. Для любого количества используемых вычислителей p справедлива следующая верхняя оценка для времени выполнения параллельного алгоритма: $T_p^{calc} < T_{\infty}^{calc} + T_1^{calc} / p$.

Для нашего примера верна следующая оценка: $T_p^{calc} < 5 + 16 / p$. Так для условного времени выполнения вычислительной модели на двух вычислителях верно $T_2^{calc} = 9 < 13$, на четырех – $T_4^{calc} = 6 < 9$.

Теорема 4. Времени выполнения алгоритма, сопоставимого с минимально возможным временем T_{∞}^{calc} , можно достичь при количестве вычислителей порядка $p \sim T_1^{calc} / T_{\infty}^{*calc}$, что с учетом теоремы 3 дает следующую оценку: $T_p^{calc} \leq 2T_{\infty}^{calc}$.

Согласно теореме 4 для выполнения вычислительной модели (рис. 5.1) потребуется порядка 16 вычислителей. Эта оценка довольно грубая, что связано с тем, что в нашем алгоритме результат одной операции может требоваться для выполнения нескольких операций. В целом оценка требуемых вычислителей для достижения минимально возможного времени выполнения вычислительной схемы тем точнее, чем меньше разница между временами T_{∞}^{*calc} и T_{∞}^{calc} .

Опираясь на эти теоремы, можно дать следующие рекомендации по выбору параллельных алгоритмов:

- 1) по теореме 1 при выборе вычислительной схемы алгоритма должен использоваться граф с минимально возможным диаметром;
- 2) по теореме 4 для параллельного выполнения целесообразное количество вычислителей определяется величиной $p \sim T_1^{calc} / T_{\infty}^{*calc}$ или максимальной шириной ярусов;
- 3) по теореме 4 время выполнения параллельного алгоритма ограничивается сверху.

Понятие ускорения и эффективности

Время выполнения последовательной программы T_1 , как уже отмечалось в (5.1), зависит от времени $T_1^{calc}(N)$, которое затрачивается собственно на вычисления, и времени обращения к памяти $T_1^{mem}(N)$.

Время выполнения программы в вычислительной среде с распределенной памятью p процессами (каждый процесс выполняется на своем вычислителе) состоит из времени $T_p^{calc}(N)$, затрачиваемого на выполнение вычислений, времени работы с памятью $T_p^{mem}(N)$ и времени $T_p^{comm}(p)$, затрачиваемого на обмены:

$$T_p(N) = T_p^{calc}(N) + T_p^{mem}(N) + T_p^{comm}(p). \quad (5.3)$$

Причем время, затраченное на вычисления и работу с памятью, как правило, уменьшается пропорционально задействованному количеству вычислителей, поскольку эти операции происходят параллельно, время обменов, следует отнести к накладным расходам:

$$T_p(N) \approx \frac{T_p^{calc}(N) + T_p^{mem}(N)}{p} + T_p^{comm}(p). \quad (5.4)$$

Время выполнения многопоточной программы в вычислительной среде с общей памятью с помощью p нитей (каждая нить выполняется на своем вычислителе) состоит из времени $T_p^{calc}(N)$, затрачиваемого на выполнение вычислений, времени работы с памятью $T_p^{mem}(N)$ и времени $T_p^{synch}(p)$, затрачиваемого на создание, уничтожение и синхронизацию нитей:

$$T_p(N) = T_p^{calc}(N) + T_p^{mem}(N) + T_p^{synch}(p). \quad (5.5)$$

Причем время, затраченное на вычисления, как правило, уменьшается пропорционально задействованному количеству нитей, поскольку вычисления происходят параллельно. Время параллельной работы p нитей с общей памятью также уменьшается, но одновременное чтение и запись несколько ухудшают степень параллелизма работы с памятью. Время создания, уничтожения и синхронизации нитей следует отнести к накладным расходам:

$$T_p(N) = \frac{T_1^{calc}(N) + \gamma(p)T_1^{mem}(N)}{p} + T_p^{synch}(p), \quad 1 < \gamma(p) < p. \quad (5.6)$$

Для описания эффективности параллельных вычислений используют следующие понятия.

Ускорением параллельного алгоритма, выполняющегося на p вычислителях, по сравнению с последовательным вариантом проведения вычислений, называется величина $S_p(N) = T_1(N)/T_p(N)$, которая в общем случае может зависеть от вычислительной сложности задачи N .

Если $S_p = p$, то говорят о линейном ускорении, но обычно ускорение на p вычислителях, как следует из формул (5.4), (5.6), оказывается меньше p . В некоторых случаях может наблюдаться и сверхлинейное ускорение ($S_p > p$). Такое поведение обычно объясняется различиями в условиях выполнения последовательной и параллельной программ. Например, данные последовательной программы не умещаются в кэше, а в результате декомпозиции по данным параллельная программа размещает их в кэше p вычислителей, что и приводит к сверхлинейному ускорению. Иногда объем вычислений самого алгоритма нелинейно зависит от количества обрабатываемых данных (например, алгоритм пузырьковой сортировки). При уменьшении количества данных, приходящихся на вычислитель, просто уменьшается суммарный объем вычислений по сравнению с тем же алгоритмом, выполняющимся последовательно.

Эффективность использования параллельного алгоритма вычислителей определяется соотношением $E_p(N) = S_p(N)/p = T_1(N)/(pT_p(N))$.

Ясно, что эффективность линейного ускорения равна единице.

Заметим, что ускорение и эффективность зависят от количества вычислителей p , вычислительной сложности N и алгоритма. Часто повышение ускорения может быть обеспечено увеличением числа вычислителей, что влечет, как правило, падение эффективности. В ряде случаев алгоритм эффективен на задачах только большой размерности (вычислительной сложности).

Говорят, что параллельный алгоритм *масштабируем*, если его эффективность постоянна для большого диапазона количества вычислителей и размерностей задач. На реальных задачах редко удастся получить идеальное масштабируемое ускорение. Обычно рост производительности программы должен быть достаточен, чтобы вообще имело смысл тратить время на распараллеливание.

Достаточно пессимистичную оценку ускорению дает закон Амдала. Дело в том, что почти в любой программе существует доля операций, которые необходимо выполнять последовательно. Пусть N_{serial} — количество операций, выполняющихся только последовательно, а $N_{parallel}$ — количество операций, которые можно выполнять параллельно; тогда $\alpha = N_{serial}/(N_{serial} + N_{parallel})$ — доля последовательных операций. Ясно, что

$0 \leq \alpha \leq 1$, крайние случаи соответствуют полностью параллельным ($\alpha = 0$) и полностью последовательным ($\alpha = 1$) программам.

Закон Амдала [9, 50]. Ускорение, получаемое при использовании параллельного алгоритма для p процессоров при данном значении α доли последовательных операций, ограничено сверху $S_p \leq 1/(\alpha + (1 - \alpha)/p)$ (рис. 5.2).

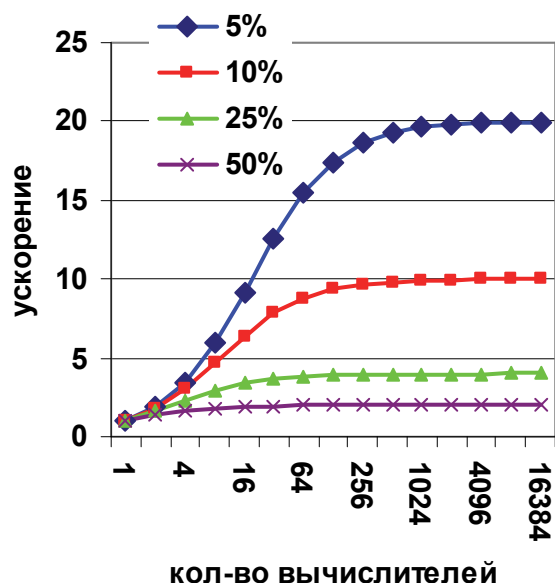


Рис. 5.2. Оценка по Амдалу максимально возможного ускорения параллельного алгоритма при разных значениях доли последовательно выполняемых операций

Например, при $\alpha = 1/10$

ускорения более чем в 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0).

Следствие закона Амдала. Для того чтобы ускорить выполнение программы в q раз, необходимо ускорить не менее чем в q раз не менее чем $(1 - 1/q)$ -ю часть программы.

Согласно следствию для получения ускорения параллельного алгоритма в 100

раз по сравнению с его последовательным вариантом необходимо получить ускорение не менее чем в 100 раз для 99,99 % кода.

Отметим, что для реальных задач плохо распараллеливаемые фазы ввода и вывода данных занимают малую часть общего времени выполнения. Кроме того, суперкомпьютеры, как правило, обеспечивают аппаратную поддержку высокоскоростного параллельного ввода. В частности, стандарт MPI-2 предоставляет специальные средства работы с файлами.

Во многих вычислительных задачах доля α последовательных операций уменьшается с ростом размерности задачи, что, в свою очередь, приводит к более оптимистичным оценкам в законе Амдала.

Оценка ускорения, использующая не долю последовательно выполняющихся операций, а время их выполнения на p вычислителях $\tau = N_{serial} / (N_{serial} + N_{parallel} / p)$, носит название закона Гюставсона – Барсиса.

Закон Гюставсона – Барсиса. Ускорение, получаемое при использовании параллельного алгоритма для p вычислителей при данном значении τ доли времени выполнения последовательных операций, ограничено сверху $S_p \leq p - (p - 1)\tau$ (рис. 5.3).

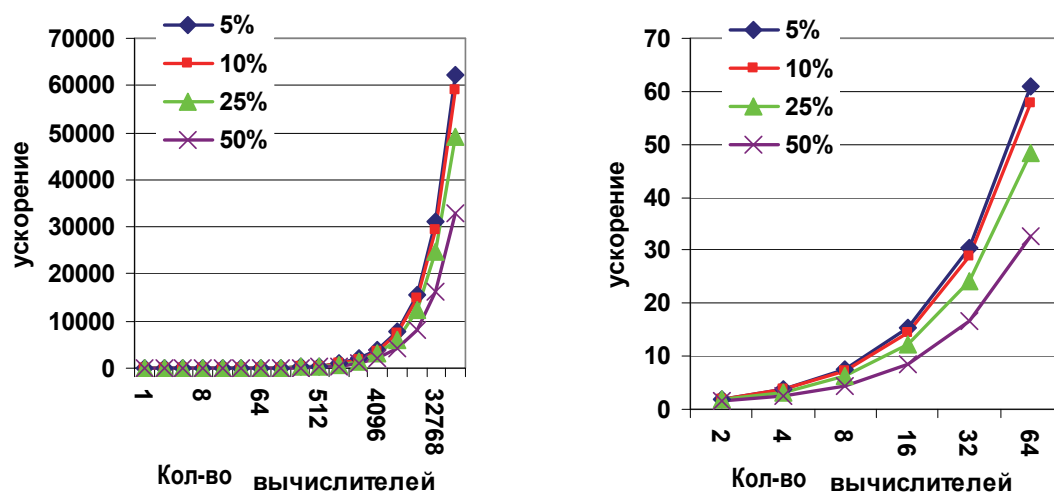


Рис. 5.3. Оценка максимально возможного ускорения параллельного алгоритма по закону Гюставсона – Барсиса при разных значениях доли времени выполнения нераспараллеленных операций

В табл. 5.1 приведены ускорения алгоритма по пессимистичной оценке Амдала и оптимистичной Гюставсона – Барсиса в зависимости от доли последовательного кода. Однако надо понимать, что никакого противоречия в оценках нет. В законе Амдала используется доля нераспараллеленных операций от общего числа операций алгоритма, а в законе Гюставсона – Барсиса – доля времени выполнения последовательной части программы.

Таблица 5.1

Доля, %	Закон	Количество вычислений							
		2	4	8	64	512	1 024	2 048	65 536
5	Амдал	1,90	3,48	5,93	15,42	19,28	19,64	19,82	19,99
	Гюставсон – Барсис	1,95	3,85	7,65	60,85	486,45	972,85	1945,65	62 259,25
10	Амдал	1,82	3,08	4,71	8,77	9,83	9,91	9,96	10,00
	Гюставсон – Барсис	1,90	3,70	7,30	57,70	460,90	921,70	1 843,30	58 982,50
25	Амдал	1,60	2,29	2,91	3,82	3,98	3,99	3,99	4,00
	Гюставсон – Барсис	1,75	3,25	6,25	48,25	384,25	768,25	1536,25	49 152,25
50	Амдал	1,33	1,60	1,78	1,97	2,00	2,00	2,00	2,00
	Гюставсон – Барсис	1,50	2,50	4,50	32,50	256,50	512,50	1024,50	32 768,50

Отметим еще раз, что в реальных масштабируемых задачах при росте размерности доля времени выполнения последовательных вычислений τ уменьшается.

Проиллюстрируем обсуждаемые в этом параграфе понятия на примере ставшей уже классической задачи поиска суммы последовательности n чисел [9, 11, 33, 35, 52, 90]. В наших рассуждениях не будем оценивать время, затрачиваемое на обращение к памяти, синхронизацию и обмен данными.

Задача 5.1. Дана последовательность чисел $\{a_i\}_{i=0}^{n-1}$. Найти $S_0^{n-1} = \sum_{i=0}^{n-1} a_i$.

1. *Последовательная схема.* Традиционный алгоритм состоит в последовательном накоплении суммы:

```
s=a[0]; for [i = 0 to n - 1] s=s+a[i];
```

и допускает только последовательное исполнение: $T_1 = T_\infty = n - 1$ (по-прежнему считаем время выполнения одной операции равным некоторой единице). Граф вычислительной схемы прост, для $n = 8$ он изображен на рис. 5.4. Здесь и далее через S_i^j обозначена частичная сумма $S_i^j = \sum_{k=i}^j a_k$.

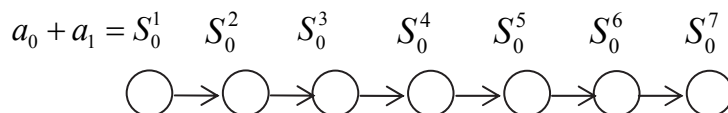


Рис. 5.4. Последовательная схема суммирования для восьми элементов

2. *Каскадная схема.* Параллелизм алгоритма суммирования становится возможен ввиду ассоциативности этой операции при использовании принципа дихотомии (гл. 1, рис. 1.15). Алгоритм выполняется в несколько этапов. Сначала находят сумму пар соседних элементов: S_0^1, S_2^3, S_4^5 и т.д. (это можно делать одновременно). На следующем шаге попарно суммируют полученные суммы: S_0^2, S_4^6 и т.д. Этапы попарного суммирования повторяются до получения окончательного результата. Граф вычислительной схемы для $n = 8$ представлен на рис. 5.5 и при наличии достаточного количества процессоров может быть выполнен параллельно по ярусам двоичного дерева.

При сохранении общего объема вычислений $T_1 = n - 1$ каскадная схема может быть выполнена за оптимальное число шагов $T_\infty(n) = T_\infty^*(n) = \log_2 n$. Без потери общности можно считать, что n есть сте-

пень двойки (в противном случае все оценки надо округлять до целого в большую сторону). При этом процессоры будут загружены неравномерно, максимальное их количество понадобится на первом ярусе – $n/2$. Таким образом, для выполнения каскадной схемы на p процессорах имеют место следующие оценки:

$$S_p \leq (n-1)/\log_2 n, \quad E_p \leq (n-1)/\left(\frac{n}{2}\log_2 n\right).$$

Отсюда, в частности, следует, что эффективность использования процессоров уменьшается при увеличении количества суммируемых значений: $E_p \rightarrow 0$ при $n \rightarrow \infty$.

3. *Модифицированная каскадная схема.* Возможна модификация каскадной схемы с целью повышения эффективности использования процессоров для больших n . Она заключается в том, что на первом этапе вычислений суммируемые значения разбиваются не на пары, а на $n/\log_2 n$ групп по $\log_2 n$ элементов; накопление суммы в каждой группе происходит последовательно, при этом вычисления в группах могут производиться параллельно на $n/\log_2 n$ процессорах. На втором этапе применяется обычная каскадная схема для суммирования результатов первого этапа. Граф вычислительной схемы для $n = 16$ представлен на рис. 5.6.

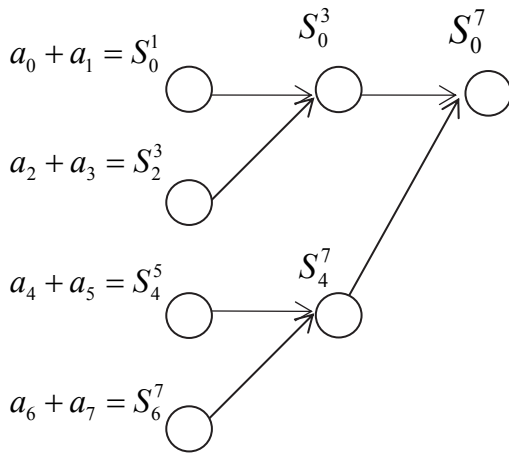


Рис. 5.5. Каскадная схема алгоритма суммирования для 8 элементов

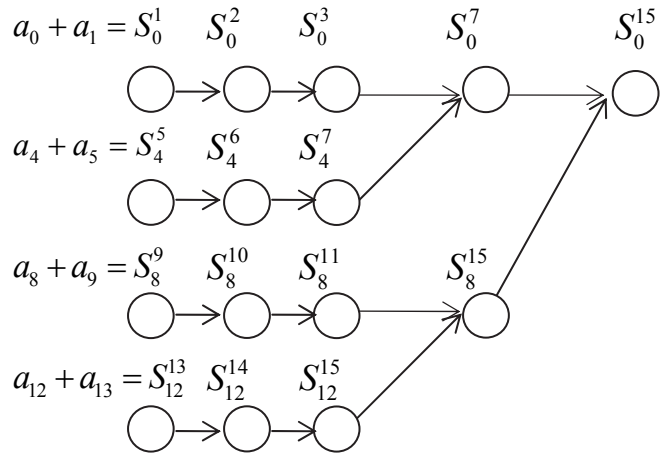


Рис. 5.6. Модифицированная каскадная схема для 16 элементов

При сохранении общего объема вычислений $T_1 = n - 1$ модифицированная каскадная схема может быть выполнена за оптимальное число шагов $T_\infty(n) = 2\log_2 n - \log_2(\log_2 n) - 1 > T_\infty^*(n) = \log_2 n$. При этом процессоры будут загружены неравномерно, максимальное их количество понадобится

на первом этапе — $n/\log_2 n$. Таким образом, для выполнения каскадной схемы на p процессорах имеют место следующие оценки:

$$S_p \leq (n-1)/2\log_2 n, \quad E_p \leq (n-1)/(2n).$$

Отсюда, в частности, следует, что хотя ускорение уменьшилось в два раза по сравнению с каскадной схемой, эффективность использования процессоров уже не стремится к нулю при увеличении количества суммируемых значений: $E_p \rightarrow 1/2$ при $n \rightarrow \infty$.

4. *Вычисление промежуточных сумм.* У последовательного алгоритма есть одно преимущество. Его применение дает все промежуточные суммы S_0^i , $i = 1, \dots, n-1$ как побочный результат, без увеличения общего количества операций (рис. 5.4). Как видно из рис. 5.5 и 5.6, ни каскадная схема, ни ее модифицированный вариант не позволяют вычислить все промежуточные суммы. Однако возможно такое построение вычислительного процесса [11, 33, 35] (рис. 5.7), что при увеличении общего объема вычислений

$$T_1(n) = n\log_2 n - \frac{1}{2}\log_2 n (\log_2 n + 1) \sim n\log_2 n$$

параллельная схема может быть выполнена за такое же оптимальное число шагов $T_\infty(n) = T_\infty^*(n) = \log_2 n$, что и каскадная схема вычисления суммы.

Схема состоит в следующем:

1) на каждом процессоре создается копия переменной для накопления суммы, которая инициализируется суммой пары элементов S_i^{i+1} , где i — номер процессора, $i = 0, \dots, n-1$;

2) далее на каждой итерации s , $s = 1, \dots, \log_2 n$ процессор с номером, превышающим $2^s - 1$, должен сложить свое значение и значение, полученное на предыдущей итерации процессором, с номером $i - 2^s$.

Ясно, что при этом процессоры будут загружены почти равномерно — на каждом i -м ярусе ($i = 1, \dots, \log_2 n$) потребуется $n - i$ процессоров. На рис. 5.7 вершины графа, соответствующие не участвующим в вычислениях процессорам, отмечены серым цветом. Эти вершины сносятся на соответствующие ярусы без изменения (пунктирные дуги) для наглядности графа.

Таким образом, для выполнения каскадной схемы на p процессорах имеют место следующие оценки:

$$S_p \leq (n-1)/\log_2 n, \quad E_p \leq 1/\log_2 n.$$

Отсюда, в частности, следует, что эффективность использования процессоров уменьшается при увеличении количества суммируемых значений:

$E_p \rightarrow 0$ при $n \rightarrow \infty$, а для улучшения эффективности вычислений может потребоваться модификация, как и в случае каскадной схемы.

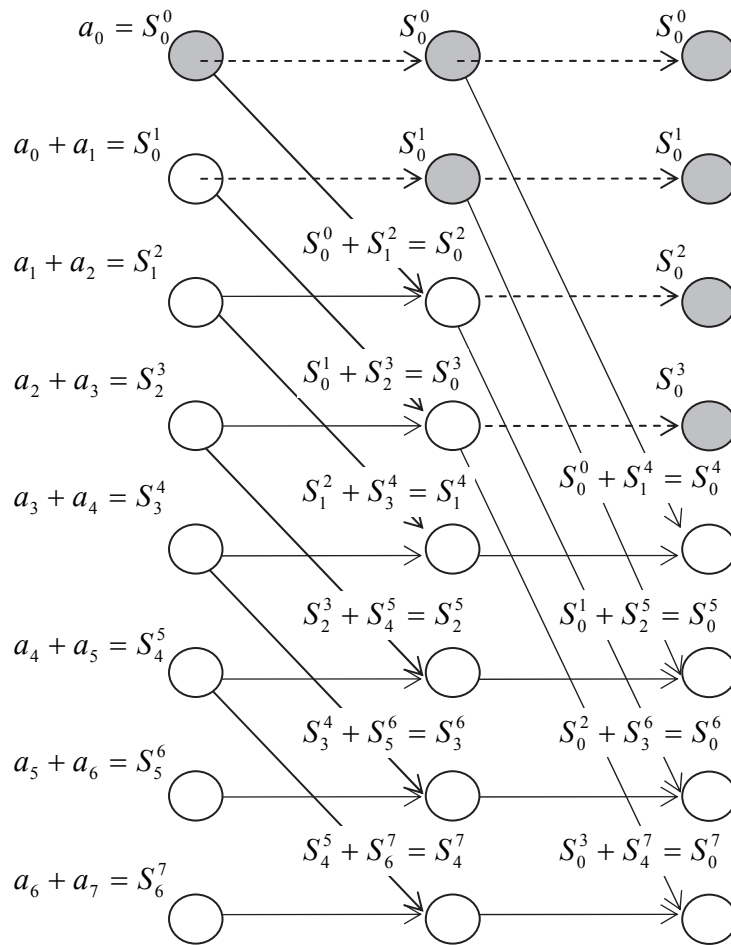


Рис. 5.7. Схема вычисления промежуточных сумм для восьми элементов

Отметим, что алгоритм суммы с накоплением порой имеет несколько иную интерпретацию, если доступны векторные операции [33, 52]. Рис. 5.7 может отражать вычисления на векторном процессоре:

1) инициализируется вектор S значениями элементов, требующих сложения;

2) каждая итерация $S = 1, \dots, \log_2 n$ заключается в формировании вектора $Q(s)$ из вектора S нециклическим сдвигом элементов на 2^s вправо с заменой слева первых значений нулями и сложении векторов S и $Q(s)$. В этом случае на рис. 5.7 серым цветом отмечены нулевые элементы вектора $Q(s)$.

Для этого алгоритма все оценки ускорения и эффективности остаются справедливыми.

Взаимодействие процессов

В предыдущем параграфе мы обсудили основные подходы к выявлению параллельных свойств выбранного последовательного алгоритма и построению эффективных собственно параллельных алгоритмов. При этом проблема физической распределенности данных и необходимости коммуникаций между процессорами не учитывалась. Между тем различные характеристики сети связи, соответствие ее физической топологии и логической топологии алгоритма имеют большое значение для ускорения и эффективности параллельного решения задачи. В этом пункте мы рассмотрим проблему распределенности данных подробнее. На практике существует небольшое количество подходов к ее решению. Рассмотрим основные из них на простом примере поиска максимума на основе введенной ранее нотации каналов и примитивов передачи сообщений.

Задача 5.2. Пусть каждый процесс знает число v . После выполнения кода все процессы должны знать глобальный по всем процессам максимум v_max .

1. *Симметричное решение, топология «полный граф»* (пример 5.2, рис. 5.8). Каждый процесс отправляет свою версию v всем процессам и, получая от них информацию, самостоятельно определяет максимум. Заметим, что если сразу производить обработку получаемой информации, то хранить все получаемые данные не обязательно.

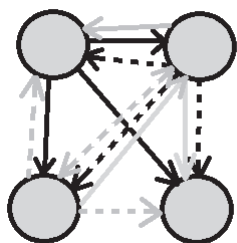


Рис. 5.8. Симметричное решение поиска максимума для четырех процессов

Пример 5.2. Псевдокод реализации симметричного решения поиска максимума

```
chan values[n](int);
//Все процессы выполняют один код
thread P (i=0; i<n){
  //Пусть v инициализирована
  int v=..., new, v_max = v;
  //Разослать мое число всем процессам
  for (j=0; ((j<n) && (j !=i)); j++){
    send values[j](v);
  }
  //Собрать значения, запоминая максимум
  for (j=1; j<n; j++){
    receive values[i](new);
    if (new > v_max) v_max = new;
  }
}
```

За счет дублирования работы по поиску максимума всеми процессами решение является симметричным (т. е. все потоки выполняют один и тот же код). Это является, пожалуй, единственным достоинством алгоритма. С точки зрения производительности такое решение при отсутствии физической связи в топологии «полный граф», позволяющей параллельно

выполнять операции `send` внутри процесса, оптимальным назвать сложно. Для его реализации на n процессах потребуется $n(n-1)$ обменов.

2. *Кольцевое решение, топология «кольцо»* (пример 5.3, рис. 5.9). При реализации этого подхода информация проходит через все процессы по кольцу. Каждый процесс сначала дожидается числа от левого соседа, сравнивает его со своим значением и отправляет правому соседу максимум, который в данном случае будет являться локальным. Глобальный максимум определится только тогда, когда информация пройдет все кольцо. Далее следует глобальный экстремум распространить по кольцу для того, чтобы этим знанием обладали все процессы.

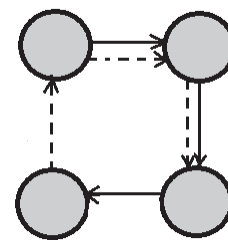


Рис. 5.9. Кольцевое решение поиска максимума для четырех процессов

Если кроме вычисления максимума процессы ничем не заняты, то такая схема вычислений, по сути, является последовательной. Все процессы занимаются поиском максимума, количество обменов равно $2(n-1)$, где n – количество процессов. Решение не является абсолютно симметричным, поскольку должен быть выделен процесс-зачинщик, который запускает конвейер. На рис. 5.9 сплошными линиями обозначены передачи локальных максимумов, пунктирными – распространение глобального максимума по кольцу.

3. *Централизованное решение, топология «звезда»* (пример 5.4, рис. 5.10). Это решение с управляющим процессом, который можно назвать плохим руководителем. Всю работу по поиску максимума он берет на себя, а рабочие процессы, сгенерировав свои числа, отправляют их управляющему процессу (по разделяемому всеми процессами каналу `values`) и ждут получения результата (в собственном канале `results[i]`, где i – номер процесса). Отметим, что при централизованной схеме решения потребуется столько же обменов, что и при кольцевой: на n процессах необходимо $2(n-1)$ обменов.

Ясно, что основным недостатком этого решения является неравномерность распределения нагрузки по процессам. Алгоритм имеет смысл применять в случае, если поиск максимума является не основной задачей и может быть выполнен управляющим процессом как фоновым. Алгоритм эффективен, если в силу логики задачи управляющий процесс все равно существует и максимум необходим только управляющему процессу (количество обменов уменьшается до $n-1$).

Алгоритмы 1–3, по сути, не распараллеливают вычисления, а решают проблему «распределенности» данных. Цель же синхронного параллельного вычисления – решить задачу быстрее, чем на одном процессоре.

Пример 5.3. Псевдокод реализации кольцевого решения поиска максимума

```

chan values[n] (int);
//Процесс - инициализатор вычислений
thread P(i=0; i<1){
  int v=...; //пусть v инициализирована
  int v_max = v; //начальное состояние глобального максимума
  //Инициировать процесс поиска глобального максимума в кольце,
  //отправив локальный максимум v следующему процессу (P[1])
  send values[1] (v_max);
  //Получить глобальный максимум от предыдущего процесса
  receive values[0] (v_max);
  //Инициировать процесс распространения глобального максимума
  //по кольцу, отправив v_max следующему процессу (P[1])
  send values[1] (v_max);
}
thread P(i=1; i<n) {
  int v=...; //пусть v инициализирована
  int v_max; //начальное состояние глобального максимума
  //Получить локальный максимум от предыдущего процесса
  receive values[i] ( v_max);
  if (v > v_max) v_max = v; //обновить локальный максимум
  //Отправить локальный максимум следующему процессу
  send values[(i + 1)%n] (v_max);
  //Получить глобальный максимум от следующего процесса
  if (i != n-1) receive values[i] (v_max);
}

```

4. *Каскадная схема решения, топология «дерево»* (пример 5.5, рис. 5.5). Схема поиска максимума повторяет каскадную схему вычисления суммы. Параллелизм алгоритма поиска максимума, как и в случае с вычислением суммы, становится возможен ввиду ассоциативности этой операции.

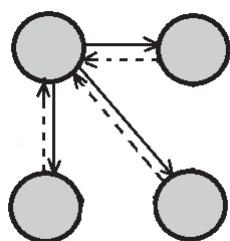


Рис. 5.10. Централизованное решение поиска максимума для четырех процессов

Код из примера 5.5 содержит каскадный алгоритм поиска максимума для 2^s процессов (необходимо выполнить s уровней, s считается известным в каждом процессе). В записи использована нотация $\text{pow}(a, b)$ для операции возведения в степень a^b . Оператор `Barrier(n)` реализует операцию коллективной синхронизации, которая завершает свое выполнение только в тот момент, когда все n процессов совершат вызов этой процедуры (см. гл. 2). Алгоритм состоит из прямого хода для поиска максимума и обратного хода для распространения знания о глобальном максимуме на

все процессы. Количество обменов такое же, как и в случае кольцевого и централизованного решений: на n процессах необходимо $2(n-1)$ обменов.

Пример 5.4. Псевдокод реализации централизованного решения поиска максимума

<pre> chan values(int), results[n](int); //Управляющий процесс thread P(i=0;i<1){ //Пусть v инициализирована int v=...; int new, v_max = v; //Собрать числа, //запоминая максимум for (j=1; j<n; j++) { receive values(new); if (new>v_max) v_max = new; } //Разослать глобальный //максимум всем процессам for (j=1; j<n; j++) send results[j](v_max); } </pre>	<pre> //Рабочий процесс thread P(i=1;i<n){ //Пусть v инициализирована int v=...; int v_max; //Послать свое число //управляющему процессу send values(v); //Получить результат //от управляющего receive results[i](v_max); } </pre>
--	--

Пример 5.5. Реализация каскадной схемы поиска максимума

```

chan values[n](int);
thread P(i=1;i<n){ //все процессы выполняют один код
  int s=...; //пусть v инициализировано
  int v=...; //количество уровней s задано
  int v_max=v; //глобальный максимум (сначала равен локальному)
  int new, m1, m2;
  //Прямой ход: поиск максимума
  //Цикл по уровням каскадной схемы
  for (j=1; j<=s; j++){
    m1=pow(2,j); m2=pow(2,j-1);
    //По номеру уровня определить партнера и произвести обмен
    if (i % m1 == m2) send values[i - m2](v_max);
    else if (i % m1 == 0) receive values[i](new);
    if (new > v_max) v_max = new; //запомнить локальный максимум
    barrier(n);
  }
  //Обратный ход: распространение глобального максимума
  //(цикл по уровням каскадной схемы в обратном направлении)
  for (j=s; s>0; s--) {
    m1=pow(2,j); m2=pow(2,j-1);
    // по номеру уровня определить партнера и произвести обмен
    if (i % m1 == 0) send values[i + m2](v_max);
    else if (i % m1 == m2) receive values[i](v_max);
    Barrier(n);
  }
}

```


Каскадная схема реализует парадигму «потребители – производители» методом дихотомии для систем с распределенной памятью и позволяет часть вычислений проводить параллельно. Хотя степень параллелизма падает с увеличением номера уровня, такая организация вычислений, как следует из предыдущего пункта (задача 5.1), наиболее эффективна [9, 11, 33, 35].

Заметим также, что каскадная схема является частным случаем общего *алгоритма пульсации*, общий код которого представлен в примере 5.6. Алгоритм пульсации используется при программировании для систем с распределенной памятью сеточных вычислений (возникающих при численном решении уравнений в частных производных или в задачах обработки изображений) и клеточных автоматов (используемых при моделировании, например, лесных пожаров или биологических систем). Алгоритм получил такое название, поскольку действие рабочих процессов напоминает работу насоса: расширение при отправке информации, сокращение при сборе новых данных, затем обработка информации и повторение цикла.

Пример 5.6. Общая схема алгоритма пульсации

```
thread Worker (i=1; i<n) {
    Декларация локальных переменных
    Инициализация локальных переменных
    Определение соседей по обходам
    //Цикл пульсации
    while (есть работа)
    {
        send значения соседям;
        receive значения от соседей;
        обновить (рассчитать) локальные значения
        Barrier(n);
    }
}
```

Задачи на поиск глобального экстремума, общей суммы и т. д. часто встречаются как составные части алгоритмов и носят название операций приведения. Языковые реализации, как правило, предоставляют высокоуровневый механизм для выполнения коллективных операций обмена и операций приведения. Как правило, суперкомпьютеры имеют аппаратную реализацию таких алгоритмов.

При коллективном взаимодействии подразумевается, что заинтересованные в нем процессы объединены в *группу*. Функция коллективного обмена для своего корректного выполнения должна быть вызвана всеми процессами группы. Определим базовый набор коллективных операций обмена в группе.

1. *Синхронизация барьером* (Barrier). Явная коллективная синхронизация *группы* процессов заключается в том, что процесс, вызывая функцию `Barrier`, приостанавливается до тех пор, пока все процессы группы не совершат вызов этой функции. Возможные способы реализации барьера обсуждались в гл. 2.

2. *Глобальные функции связи* включают различные способы обмена данными в группе (рис. 5.11).

2.1. *Рассылка данных* – передача данных от одного процесса группы ко всем процессам группы – `Broadcast(root, ...)`. Процесс-инициатор операции принято называть корневым (`root`).

2.2. *Обобщенная рассылка данных* от одного процесса группы ко всем процессам группы – `Scatter(root, ...)`. Эта операция подразумевает распределение набора данных корневого процесса по всем процессам группы, т. е. каждый процесс получает от корневого свою порцию данных.

2.3. *Сбор данных* от всех процессов группы к одному процессу этой группы – `Gather(root, ...)`. В этом случае корневым называется процесс-получатель: он собирает данные от всех процессов, размещая их в своем буфере. Операция является в некотором смысле обратной к `Scatter()`.

2.4. *Обобщенный сбор данных* – получение всех собираемых данных в каждом процессе группы – `Allgather()`. Эта операция обобщает `Gather()`, однако является симметричной, т. е. не имеет корневого процесса-инициатора.

2.5. *Совмещенные рассылка и сбор данных* (`Scatter/Gather`) от всех процессов группы всеми процессами группы – `Alltoall()`. Эта операция является наиболее общей операцией коллективного обмена данными.

На рис. 5.11 проиллюстрировано выполнение глобальных функций связи для шести процессов. В каждой клетке представлены локальные данные в одном процессе. Если операция подразумевает корневой процесс, то его роль на рисунке выполняет 0-й процесс. Например, при выполнении операции `broadcast` данные A_0 передает только процесс с номером 0, а другие процессы принимают эти данные.

3. *Глобальные операции приведения* (рис. 5.12). Эти операции носят также название операций редуцирования и совмещают коллективный обмен с выполнением над данными таких операций, как сумма, произведение, поиск максимума или минимума, логические операции по всем процессорам.

3.1. Редуцирование, при котором *результат возвращается только одному процессу группы* – `Reduce(root, ...)`, – выбранному в качестве корневого.

3.2. Редуцирование, при котором результат возвращается всем процессам группы, – `Allreduce()`.

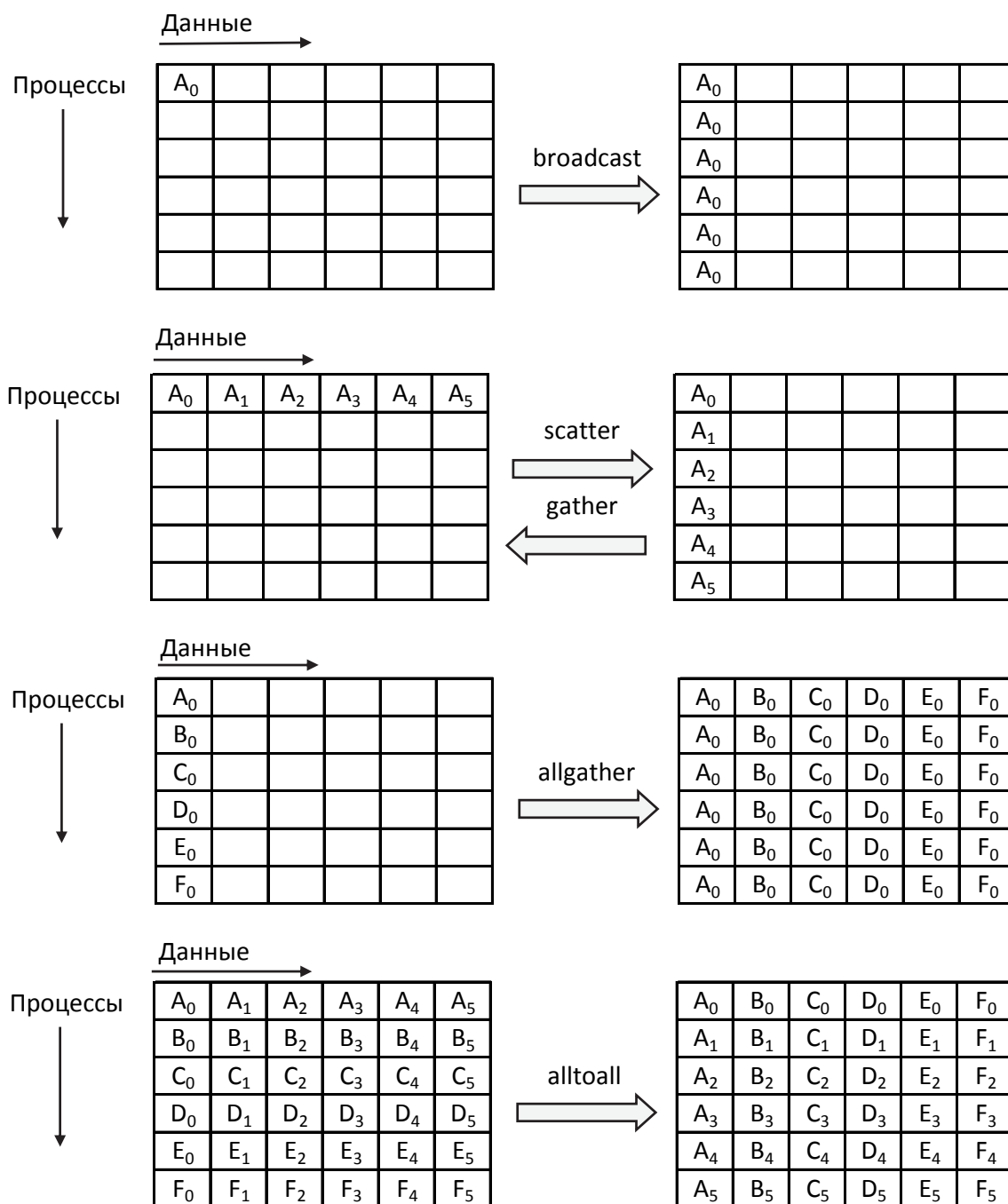


Рис. 5.11. Глобальные функции связи для группы из шести процессов (процесс с номером 0 считается корневым)

3.3. Последовательное выполнение операции редуцирования `reduce()` и операции обобщенной рассылки `Scatter()` – `ReduceScatter()`.

3.4. *Развертка данных* по всем процессорам группы – `Scan()`. Эта операция напоминает алгоритм вычисления частичных сумм с распределением их по процессам (рис. 5.12).

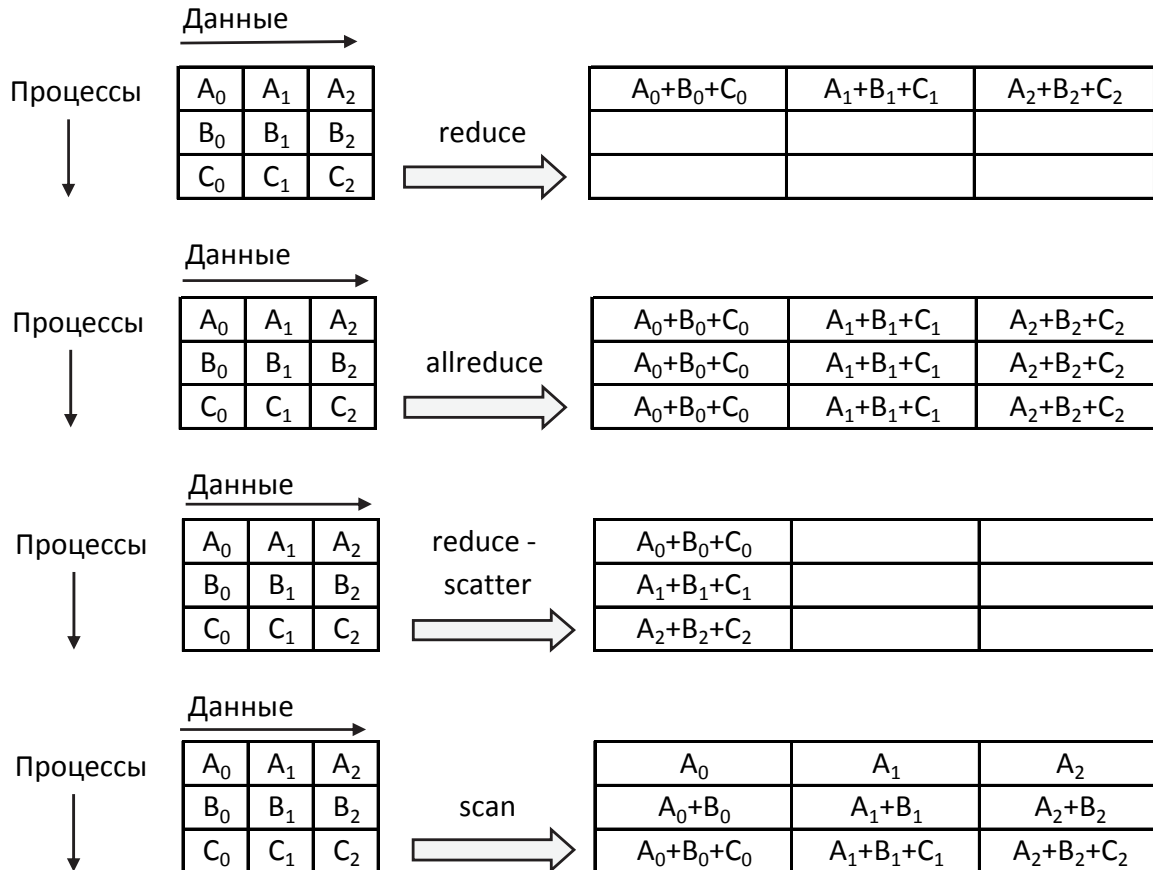


Рис. 5.12. Глобальные операции приведения для группы из трех процессов

Оценки коммуникационной трудоемкости параллельных алгоритмов

Кроме участков кода, которые необходимо выполнять последовательно, у параллельной программы есть еще три источника накладных расходов: 1) создание и сопровождение процессов; 2) взаимодействие процессов; 3) синхронизация процессов. Все эти расходы взаимосвязаны – и часто уменьшение влияния одного из них приводит к увеличению другого. Рассмотрим стандартные способы уменьшения этих расходов.

1. Для уменьшения расходов, связанных с *сопровождением* процессов, можно создавать строго один процесс на каждом вычислителе. Однако это может увеличить простои вычислителя или усложнить задачу балансировки нагрузки.

2. *Взаимодействие процессов* с помощью обмена сообщениями – это накладные расходы на инициализацию операции обмена (как у отправите-

ля, так и у получателя) и расходы, связанные с транспортировкой сообщения по Сети. Поскольку с точки зрения отправки одного сообщения эти расходы неустраняемы, то единственный способ их уменьшения – минимизация количества обменов и выполнение операции обмена, если это возможно, в фоновом режиме.

3. *Синхронизация* при программировании в ВС с распределенной памятью происходит явно при выполнении барьеров и неявно при передаче сообщений. Использование неблокирующих режимов обмена, отправка сообщений пакетами, дополнительные локальные вычисления каждым процессом вместо глобальных операций (если это возможно) – это способы уменьшения накладных расходов на синхронизацию.

Обычно среда выполнения операции обмена характеризуется следующими понятиями.

Латентность – время между инициированием передачи данных при отправке и прибытия первого байта при приеме.

Латентность часто зависит от длины посылаемых сообщений. Ее значение может изменяться в зависимости от того, послано ли большое количество маленьких сообщений или нескольких больших сообщений.

Латентность характеризуется тремя параметрами: $t_{prepare}$ – время подготовки данных; t_{serv} – время передачи служебных данных; t_{byte} – время, необходимое для передачи одного байта.

Пропускная способность – это величина, обратная времени, необходимому для передачи одного байта $1/t_{byte}$.

Пропускная способность обычно выражается в Мб/с (Гб/с). Пропускная способность важна, когда передаются сообщения больших размеров.

Приведем базовые оценки трудоемкости обмена данными для кластерных систем [10, 90].

Топология сети кластера редко представлена *полным графом*, обычно имеются определенные ограничения на одновременность выполнения коммуникационных операций. Использование *метода передачи пакетов* (реализуемого, как правило, на основе протокола TCP/IP) является характерным для выполнения коммуникационных операций.

Базовой оценкой времени передачи сообщения размером m байт является модель Хокни [27, 49, 65, 90]:

$$t_{db} \sim t_{prepare} + mt_{byte}, \quad (5.7)$$

или с учетом времени на передачу служебной информации:

$$t_{db} \sim t_{prepare} + mt_{byte} + t_{serv}. \quad (5.8)$$

Однако если сообщение достаточно велико и составляет несколько пакетов, то эта оценка требует уточнения, поскольку не учитывает зависи-

мость $t_{prepare}$ и t_{serv} от количества пакетов. Оценки, описанные ниже, основаны на модели LogGP [57, 90].

Пусть V_{\max} определяет максимальный размер пакета, который может быть доставлен в Сеть (например, для ОС MS Windows в Сети Fast Ethernet $V_{\max} = 1500$ байт); V_{serv} определяет объем служебных данных в каждом из пересылаемых пакетов (для протокола TCP/IP, ОС MS Windows и Сети Fast Ethernet $V_{serv} = 8$ байт). Тогда количество пакетов, на которое разбивается сообщение, можно характеризовать следующей величиной: $n \sim \lceil m / (V_{\max} - V_{serv}) \rceil$. Здесь и далее скобками $\lceil \rceil$ обозначено округление числа до ближайшего большего целого.

Пусть также $t_{prepare}^0$ характеризует аппаратную составляющую латентности, которая зависит от параметров используемого сетевого оборудования, а значение $t_{prepare}^1$ задает время подготовки одного байта данных для передачи по Сети. Предположим, что латентность увеличивается линейно в зависимости от объема передаваемых данных: $t_{prepare} = t_{prepare}^0 + kt_{prepare}^1$, причем подготовка данных для передачи второго и всех последующих пакетов может быть совмещена с пересылкой по Сети предшествующих пакетов, следовательно, латентность ограничена сверху:

$$t_{prepare} \sim t_{prepare}^0 + (V_{\max} - V_{serv})t_{prepare}^1.$$

Учтем также, что с увеличением количества пакетов растет время на пересылку служебной информации t_{serv} . В результате вместо оценки (5.7) можно записать более точную оценку времени, требующегося для передачи сообщения размером m байт [13]:

$$t_{db} \sim \begin{cases} t_{prepare}^0 + mt_{prepare}^1 + (m + V_{serv})t_{byte}, & n = 1; \\ t_{prepare}^0 + (V_{\max} - V_{serv})t_{prepare}^1 + (m + nV_{serv})t_{byte}, & n > 1. \end{cases} \quad (5.9)$$

На сайте [90] представлены данные вычислительных экспериментов с использованием библиотеки MPI по изучению времени, затрачиваемого на передачу сообщений большого диапазона длин, в Сети многопроцессорного кластера Нижегородского госуниверситета им. Н. И. Лобачевского. Приведенные данные свидетельствуют об очень хорошей точности предсказаний с помощью оценки (5.9) для сообщений любой длины. Оценки (5.7) и (5.8) дают правдоподобные результаты при отправке большого количества сообщений большой длины. С учетом того, что они требуют минимум информации о характеристиках среды, их можно использовать для грубых оценок времени коммуникаций в параллельной программе.

5.3. Реализация базовых алгоритмов вычислительной математики

Многие алгоритмы вычислительной математики включают в себя как составные части умножение матрицы на вектор, произведение матриц (которое тоже можно интерпретировать как произведение матрицы на пул векторов) или решение СЛАУ. Обсудим кратко подходы к распараллеливанию этих алгоритмов. Детальное рассмотрение параллельной реализации основных алгоритмов можно найти в работах [11, 18, 19, 33, 35, 70].

Способы балансировки вычислительной нагрузки

Декомпозиция по данным предполагает распределение по процессам не задач, а данных. Причем зачастую от удачного способа декомпозиции данных зависит эффективность параллельного алгоритма. Например, при обычной декомпозиции матрицы непрерывными полосами метод Гаусса для решения СЛАУ является практически последовательным, однако его внутренний параллелизм можно заметно улучшить, изменив способ распределения матрицы по процессам.

Без потери общности проблему балансировки нагрузки можно сформулировать следующим образом.

Задача статической (априорной) балансировки вычислительной нагрузки. Пусть дано n элементов данных $\{a_i\}_{i=0}^{n-1}$, причем объем вычислений, необходимых для обработки одного элемента, одинаков для всех i . Будем считать, что каждый элемент a_i далее дробить в рамках выбранного алгоритма нецелесообразно; поэтому можно считать, что дан массив n элементов $a[n]$. Пусть для обработки данных доступно $P \ll n$ процессов, каждый из которых выполняется на отдельном вычислительном устройстве (процессоре, ядре и т. п.). Необходимо распределить данные по процессам таким образом, чтобы вычислительная нагрузка на каждый процесс была одинаковой (или как можно более равномерной).

Поскольку в общем случае количество элементов n не кратно количеству процессов P , то равномерного распределения нагрузки добиться чаще всего невозможно. Оптимальным вариантом в общем случае будет такое распределение элементов по процессам, при котором некоторые процессы получают для обработки на один элемент больше, чем остальные. Этот случай рассматривался, в частности, в ряде примеров гл. 4.

Действительно, пусть $\text{int } c1=n/P; \text{ int } c2=n\%P$ (n — количество распределяемых элементов, P — количество процессов в группе). Тогда $P-c2$ процессов получают по $c1$ элементов, а $c2$ процессов — по $c1+1$ элемент, что в нотации языка Си можно записать в следующем виде:


```
int local_n = n/P + (myID < n%P);
```

Здесь `myID` – номер процесса в группе, `local_n` – количество элементов, которое он получит.

Существует, по крайней мере, три схемы распределения элементов данных по процессам [33].

1. *Блочная схема* предполагает распределение данных непрерывными блоками по `local_n` элементов на процессор (рис. 5.13, а; 5.14, а). Код примера 5.7 содержит фрагмент, определяющий начальный `beg` и конечный `fin` адреса блока элементов массива данных, который выделяется для обработки процессу с номером `myID` при блочной схеме распределения нагрузки.

Пример 5.7. Блочная схема распределения вычислительной нагрузки

```
...
int n, P, myID, c1=n/P, c2=n%P;
//Адреса начала и конца блока элементов,
//обрабатываемого процессом myID
int beg, fin;
if (myID<c2){ //первым c2 процессам по c1+1 элементу
    beg=myID*(c1+1);
    fin= (myID+1)*(c1+1)-1;
}
else if (myID>=c2){ //остальным P-c2 процессам по c1 элементу
    //сдвиг в адресах на c2,
    //поскольку уже распределено c2*c1+c2 элементов
    beg=myID*c1+c2;
    fin= (myID+1)*c1+c2-1;
}
...
```

2. *Циклическая блочная схема* предполагает распределение массива данных небольшими блоками по `B` элементов на процесс по кругу (рис. 5.13, б; 5.14, б). В этом случае каждый процесс будет обрабатывать массив блоков элементов и требуется уже вычислить массив начальных и конечных адресов блоков, назначенных процессу с номером `myID`. Причем при балансировке нагрузки кроме параметров `c1` и `c2` потребуются еще два параметра: `int c3=c1/B; int c4=c1%B`. Параметр `c3` показывает минимальное количество полных блоков, которое будет обрабатывать каждый процесс. Параметр `c4` показывает минимальное количество элементов в последнем `(c3+1)`-м неполном блоке, которое будет обрабатывать `P-c2` процессов; соответственно, `c2` процессов будут обрабатывать `c4+1` элемента в последнем блоке (отметим, что блок, возможно, будет в этом случае полным).

Количество блоков, распределяемое на процесс, определяется значениями параметров c_2 и c_4 :

- если $c_4 \neq 0$, то все процессы обрабатывают c_3+1 блока;
- если $c_2 \neq 0$ и $c_4=0$, то первые c_2 процессов обрабатывают c_3+1 , а оставшиеся – c_3 блока;
- если $c_2=0$ и $c_4=0$, то каждый процесс обрабатывает строго c_3 блока.

Таким образом, количество блоков, приходящихся на процесс с номером $myID$, вычисляется по формуле

$$\text{int block_n} = c_3 + (c_4 > 0) + (!c_4 \ \&\& \ myID < c_2); \quad (5.10)$$

Код примера 5.8 содержит фрагмент, определяющий в самом общем случае массив начальных $beg[i]$ и конечных $fin[i]$ адресов блоков элементов данных, которые назначаются для обработки процессу с номером $myID$ при циклической блочной схеме распределения нагрузки небольшими непрерывными блоками по B элементов. В примере массивы beg и fin двумерные, поскольку заполняются для каждого процесса $myID=0, 1, \dots, P-1$. Дадим краткие пояснения помеченных в комментариях участков кода.

(1) Все процессы получают, по крайней мере, по c_3 блока, поэтому адреса первых c_3 блоков заполняются в цикле. Поскольку размер каждого i -го блока равен B и за один проход внутреннего цикла заполняется P блоков, то у i -го блока накапливается смещение всех адресов на величину $i * B * P$.

(2) Последний дополнительный, возможно неполный, блок, как следует из формулы (5.10), появляется в случае, если хотя бы один из параметров c_2 или c_4 не равен нулю. Адреса начала и конца этого блока следует заполнить отдельно.

(3) Первые c_2 процесса получают дополнительный блок размером c_4+1 . Если $c_2=0$, то все процессы имеют дополнительный блок, состоящий из c_4 элементов, их адреса заполняются в блоке `else`, поскольку ни один $myID$ не удовлетворяет условию $myID < c_2$. Смещение всех адресов равно $c_3 * B * P$. К этому моменту распределено по c_3 блокам по B элементов на P процессов.

(4) Оставшиеся процессы получают по c_4 элемента. Проверка на неравенство нулю c_4 является обязательной, поскольку в противном случае дополнительного блока у этих процессов нет. При $c_2=0$ под условие (4) попадают все процессы. Смещение всех адресов равно $c_3 * B * P + c_2$. К этому моменту распределено по c_3 блока по B элементов на P процессов, а также c_2 процесса получили на 1 элемент больше, чем c_4 .

3. *Циклическая блочная схема с отражением* предполагает распределение массива данных небольшими блоками по B элементов на процесс «веретеном» (рис. 5.13, в; 5.14, в).

Пример 5.8. Циклическая блочная схема распределения вычислительной нагрузки по блокам

```

#include <stdio.h>
#include <stdlib.h>
#define N (int)72
#define P (int)5
#define B (int)4
int main( int argc, char **argv ) {
    int myID,i,c1,c2,c3,c4,n,n_block;
    //Массивы адресов начала и конца блока элементов,
    //обрабатываемых процессом myID
    int **beg, **fin;
    c1=N/P; c2=N%P; c3=c1/B; c4=c1%B;
    printf("N=%d; P=%d; B=%d\n",N,P,B);
    printf("c1=%d; c2=%d; c3=%d; c4=%d\n",c1,c2,c3,c4);
    //DYNAMIC memory allocation
    beg=(int**)malloc(sizeof(int*)*P);
    if(!beg){printf("error in malloc: beg\n");exit(1);}
    fin=(int**)malloc(sizeof(int*)*P);
    if(!fin){printf("error in malloc: fin\n");exit(1);}
    for (myID=0;myID<P;myID++){
        n_block=c3+(c4>0)+(!c4&&myID<c2);
        beg[myID]=(int*)malloc(sizeof(int)*n_block);
        if(!beg[myID]){
            printf("error in malloc: beg %u\n",myID);exit(1);}
        fin[myID]=(int*)malloc(sizeof(int)* n_block);
        if(!fin[myID]){
            printf("error in malloc: fin %u\n",myID);exit(1);}
    }
    //Определение адресов начала и конца блоков элементов
    //в глобальной нумерации массива данных
    for (myID=0;myID<P;myID++){
        for (i=0;i<c3;i++){ //см. пояснение в тексте (1)
            beg[myID][i]=i*B*P+myID*B;
            fin[myID][i]=i*B*P+(myID+1)*B-1;
        }
        if (c2!=0 || c4!=0){ //см. пояснение в тексте (2)
            if (myID<c2){ //см. пояснение в тексте (3)
                beg[myID][c3]=c3*B*P+myID*(c4+1);
                fin[myID][c3]=c3*B*P+(myID+1)*(c4+1)-1;
            }
            else if((myID>=c2)&&(c4!=0)){ //см. пояснение в тексте
(4)
                beg[myID][c3]=c3*B*P+c2+myID*c4;
                fin[myID][c3]=c3*B*P+c2+(myID+1)*c4 -1;
            }
        }
    }
    return 0;
}

```

myID	0					1					2			
a[i]	0	1	2	3	4	5	6	7	8	9	10	11	12	13

a

myID	0		1		2		0		1		2		0	1
№ блока	0		0		0		1		1		1		2	2
a[i]	0	1	2	3	4	5	6	7	8	9	10	11	12	13

б

myID	0		1		2		2		1		0		0	1
№ блока	0		0		0		1		1		1		2	2
a[i]	0	1	2	3	4	5	6	7	8	9	10	11	12	13

в

Рис. 5.13. Пример распределения по трем процессам массива из 14 элементов.
Схемы распределения: *a* – блочная; *б* – циклическая блочная;
в – циклическая блочная с отражением

myID	0	1	2	3	4
beg	0	10	20	30	39
fin	9	19	29	38	47

a

№ блока	i=0		i=1		i=2		i=3	
myID	beg	fin	beg	fin	beg	fin	beg	fin
0	0	2	15	17	30	32	45	45
1	3	5	18	20	33	35	46	46
2	6	8	21	23	36	38	47	47
3	9	11	24	26	39	41		
4	12	14	27	29	42	44		

б

№ блока	i=0		i=1		i=2		i=3	
myID	beg	fin	beg	fin	beg	fin	beg	fin
0	0	2	27	29	30	32		
1	3	5	24	26	33	35		
2	6	8	21	23	36	38	47	47
3	9	11	18	20	39	41	46	46
4	12	14	15	17	42	44	45	45

в

Рис. 5.14. Пример балансировки нагрузки с параметрами
 $n = 48, P = 5, B = 3; c_1 = 9, c_2 = 3, c_3 = 3, c_4 = 0$.
Схемы распределения: *a* – блочная; *б* – циклическая блочная;
в – циклическая блочная с отражением

Количество блоков, распределяемых на процесс, как и в случае циклической блочной схемы распределения, определяется значениями параметров c_2 и c_4 , но больше элементов для обработки получают первые c_2 процессов, если количество полных блоков c_3 – четное, и последние c_2 процесса, если c_3 – нечетное. Учтем это, изменив в формуле (5.10), вычисляющей количество блоков, распределяемых на процесс, только последний член:

```
int block_n=c3+(c4>0)+(!c4&&(myID+(c3%2)*(P-1-2*myID))<c2); (5.11)
```

В случае когда параметр c_3 нечетный, то последние c_2 процесса получают дополнительный элемент, соответственно их номера вычисляются с конца $P-1-myID$, что и учтено в поправке в последнем члене (5.11).

Код примера 5.9 содержит фрагмент, определяющий в самом общем случае массив начальных $beg[i]$ и конечных $fin[i]$ адресов блоков элементов данных, которые назначаются для обработки процессу с номером $myID$ при циклической блочной с отражением на схеме распределения нагрузки небольшими непрерывными блоками по B элементов.

Код примера 5.9 отличается от кода примера 5.8 следующим.

1. Расчет количества блоков, распределяемых на процесс, выполняется по формуле (5.11).

2. Изменен порядок расчета адресов. Для удобства внешний цикл идет по номерам блоков, а внутренний – по номерам процессов. Как следствие, расчет последнего блока вынесен в отдельный цикл по номерам процессов.

3. С учетом четности c_3 введена поправка в вычисления индекса массива, соответствующего номеру процесса.

Пример 5.9. Циклическая блочная схема распределения вычислительной нагрузки по блокам

```
#include <stdio.h>
#include <stdlib.h>
#define N (int)72
#define P (int)5
#define B (int)4
int main( int argc, char **argv ) {
    int myID,i,c1,c2,c3,c4,n,n_block;
    //Массивы адресов начала и конца блока элементов,
    //обрабатываемых процессом myID
    int **beg, **fin;
    c1=N/P; c2=N%P; c3=c1/B; c4=c1%B;
    printf("N=%d; P=%d; B=%d\n",N,P,B);
    printf("c1=%d; c2=%d; c3=%d; c4=%d\n",c1,c2,c3,c4);
    // DYNAMIC memory allocation
    beg=(int**)malloc(sizeof(int*)*P);
```

Пример 5.9 (окончание). Циклическая блочная схема распределения вычислительной нагрузки по блокам

```
if(!beg){printf ("%s\n","error in malloc: beg");exit(1);}
fin=(int**)malloc(sizeof(int*)*P);
if(!fin){printf ("%s\n","error in malloc: fin");exit(1);}
for (myID=0;myID<P;myID++){
    // см. пояснения в тексте (1)
    n_block=c3+(c4>0)+(!c4&&(myID+(c3%2)*(P-1-2*myID)<c2));
    beg[myID]=(int*)malloc(sizeof(int)*n_block);
    if(!beg[myID]){
        printf ("error in malloc: beg %u\n",myID);exit(1);}
    fin[myID]=(int*)malloc(sizeof(int)* n_block);
    if(!fin[myID]){
        printf ("error in malloc: fin %u\n",myID);exit(1);}
}
//Определение адресов начала и конца блоков элементов
//в глобальной нумерации массива данных
for (i=0;i<c3;i++) //см. пояснение в тексте (2)
    for (myID=0;myID<P;myID++){
        //см. пояснение в тексте (3)
        beg[myID+(i%2)*(P-1-2*myID)][i]=i*B*P+myID*B;
        fin[myID+(i%2)*(P-1-2*myID)][i]=i*B*P+(myID+1)*B-1;
    }
for (myID=0;myID<P;myID++){ //см. пояснение в тексте (2)
    if (c2!=0 || c4!=0){
        if (myID<c2){ //см. пояснение в тексте (3)
            beg[myID+(c3%2)*(P-1-2*myID)][c3]=c3*B*P+myID*(c4+1);
            fin[myID+(c3%2)*(P-1-2*myID)][c3]=
                c3*B*P+(myID+1)*(c4+1)-1;
        }
        else if((myID>=c2)&&(c4!=0)){ //см. пояснение в тексте
(3)
            beg[myID+(c3%2)*(P-1-2*myID)][c3]=c3*B*P+c2+myID*c4;
            fin[myID+(c3%2)*(P-1-2*myID)][c3]=
                c3*B*P+c2+(myID+1)*c4 -1;
        }
    }
}
... // вывод на экран
return 0;
}
```

Отметим, что задача балансировки в общем случае гораздо сложнее, поскольку вычислительный объем, приходящийся на каждый процесс, может меняться со временем. Например, из-за пространственной неравномерности и временной изменчивости распределения какой-либо вычисляемой величины количество операций с одной и той же частью массива

в различные моменты времени может существенно отличаться, что делает проблему динамической балансировки нагрузки сложной и очень важной задачей.

Умножение матрицы на вектор

Задача 5.3. Пусть дана матрица A размерностью $m \times n$ и вектор \mathbf{x} размерностью n :

$$A = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0n-1} \\ a_{10} & a_{11} & \dots & a_{1n-1} \\ \dots & \dots & \dots & \dots \\ a_{m-10} & a_{m-11} & \dots & a_{m-1n-1} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix}.$$

Необходимо найти произведение $A\mathbf{x}$.

Существует два основных способа параллельного представления алгоритма умножения матрицы на вектор [33, 35].

Алгоритм 1 (скалярных произведений). Матрица A разрезается по строкам $\{a_i\}$ (группам строк). Тогда каждый элемент результирующего вектора суть скалярное произведение строки $\{a_i\}$ на вектор \mathbf{x} :

$$\mathbf{c} = A\mathbf{x} = \begin{pmatrix} (a_0, \mathbf{x}) \\ (a_1, \mathbf{x}) \\ \dots \\ (a_{m-1}, \mathbf{x}) \end{pmatrix}, \quad (\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{n-1} x_i y_i.$$

Произведения можно вычислять параллельно. В результате алгоритм для реализации первого способа кратко можно записать следующим образом.

1. *Декомпозиция данных.* Матрица разрезается на горизонтальные полосы. Каждый процесс должен хранить полосу матрицы A и целый вектор \mathbf{x} .

1.1. Если все данные сосредоточены на одном процессе, то на этом этапе их необходимо распределить по всем процессам, используя блочную схему балансировки нагрузки и глобальные функции связи $\text{Scatter}(A)$, $\text{Broadcast}(\mathbf{x})$.

1.2. Для больших по размеру матриц не выгодно хранить все данные на одном процессе, можно написать последовательную программу-препроцессор, которая подготовит, например, файлы с исходными данными для каждого процесса. Тогда на этом этапе каждый процесс должен считать данные из своего файла. Возможно также, что каждый процесс, исходя из своего номера и общего количества процессов, сгенерирует свою часть данных.

1.3. Наконец возможно, что матрица распределена по процессам на предыдущем этапе алгоритма. Если распределение сильно неравномерное, то для улучшения параллелизма необходимо провести перегруппировку данных.

2. *Вычисления.* Параллельно вычисляются все скалярные произведения (a_i, \mathbf{x}) .

3. *Сборка результата.* Если необходимо, то результат собирается на нулевом процессе с помощью глобальной функции связи `Gather(c)`.

Отметим, что при очень больших n оправдано хранить вектор \mathbf{x} разрезанным на части (лучше согласованно с полосами матрицы) и организовать передачу по кольцу частей вектора. Следует понимать, что это приведет к увеличению количества обменов.

Пример 5.10 содержит псевдокод, реализующий алгоритм для случая $P = m = n$, где P – количество процессов в группе. При объявлении двумерного массива первый параметр соответствует количеству строк, а второй – количеству столбцов. Предполагается, что каждый процесс группы знает свой номер `myID` в группе и общее количество процессов в группе n ; все данные вначале сосредоточены на нулевом процессе, матрица распределяется по процессам построчно, а вектор поэлементно.

В используемой здесь и далее нотации функций коллективных взаимодействий первый аргумент соответствует буферу, из которого производится рассылка, второй аргумент определяет корневой процесс, а третий – буфер приема в принимающем процессе. Соответственно во всех процессах, кроме корневого, первый параметр игнорируется в вызовах функции `Scatter()`, а последний – в вызовах функции `Gather()`. Символ «*» в размерности массива-аргумента в глобальных функциях связи означает, что по этому направлению *не* проводится декомпозиция или сборка массива, т. е., например, при декомпозиции по строкам «*» ставится в индексе, соответствующем столбцам. Обычно реализация глобальных функций связи предполагает неявный барьер в начале и конце операции.

Отметим также, что в примере 5.10 управляющий процесс занимается инициализацией и распределением начальных данных, сборкой и обработкой результата как бы в нагрузку, поскольку включен в процесс вычислений, наряду с остальными. Альтернативой этому может быть создание отдельного управляющего процесса, занимающегося исключительно сопровождением данных.

И первый, и второй варианты имеют ряд преимуществ и недостатков для эффективности параллельного алгоритма. С одной стороны, выделение холостого с точки зрения вычислений процесса несколько снизит время выполнения программы (ведь в противном случае вычислением управляющий процесс сможет заняться только после распределения данных,

а сбором результатов – только после окончания своих вычислений), кроме того, код такой программы более структурирован. С другой стороны, наличие лишнего процесса в знаменателе снизит показатель эффективности программы. К вопросу о роли управляющего процесса мы еще вернемся в гл. 6 при обсуждении технологии MPI.

Пример 5.10. Умножение матрицы на вектор. Псевдокод алгоритма I с круговым конвейером

```
chan values_x[n](int);
thread MatrixVectorProduct1 (myID=0; myID < n){
  if (myID == 0) {
    double aa[n,n], xx[n]; //исходные матрица A и столбец x
    double cc[n];          //результатирующий вектор c
    инициализировать a и x;
  }
  double a[n], x; //строка i матрицы A и один элемент вектора x
  double c=0; //элемент вектора-результата c
  int j, k, next_p, prev_p; //номера соседних процессов
  //Определение по своему номеру myID номеров соседей
  if (myID == n-1) next_p = 0; else next_p = myID + 1;
  if (myID == 0) prev_p = n-1; else prev_p = myID - 1;
  //Получить строку myID матрицы A от корневого процесса
  Scatter(aa[][*], root=0, a[*]);
  //Получить элемент myID вектора x от корневого процесса
  Scatter(xx[], root=0, x);
  //Пустить по кругу элементы вектора x и вычислить c[myID]
  for (j=0; j<n-1; j++) {
    k=(myID+n-j)%n
    //Вычислить с накоплением часть элемента c[myID]
    c=c+a[k]*x;
    //Отправить мой элемент вектора x следующему процессу
    send values_x[next_p](x);
    //Получить новый элемент вектора x от предыдущего процесса
    receive values_x[myID](x);
  }
  //Вычислить оставшуюся часть c[myID]
  k=(myID+1)%n; c=c+a[k]*x;
  Barrier
  //Отправить результат корневому процессу
  Gather(c, root=0, cc[]);
  Barrier
  if (myID == 0) вывести результат, который теперь в векторе
  cc;
}
```

Алгоритм II (линейная комбинация). Этот алгоритм предполагает, что матрица A разрезается по столбцам $\{a_j\}$ (полосам столбцов). Тогда

вычисление произведения суть вычисление линейной комбинации системы из n векторов $\{a_j\}$:

$$\mathbf{c} = A\mathbf{x} = \sum_{j=1}^n x_j \mathbf{a}_j$$

При этом j -й процесс будет хранить *все m частичных сумм* для каждого элемента результирующего вектора, а для вычисления результата следует выполнить суммирование по процессам. В результате алгоритм для реализации второго способа кратко можно записать следующим образом.

1. *Декомпозиция данных.* Матрица A разрезается на вертикальные полосы. Каждый процесс должен хранить полосу матрицы A и соответствующее ширине своей полосы количество элементов вектора \mathbf{x} (`Scatter(A)`, `Scatter(x)`). Все соображения, высказанные в пп. 1.1–1.2 предыдущего алгоритма, остаются в силе.

2. *Вычисления.* Параллельно вычисляются все частичные суммы линейной комбинации.

3. *Сборка результата.* Результирующий вектор \mathbf{c} с суммированием собирается на нулевом процессе (`Reduce(...sum)`) или на каждом процессе (`Allreduce(...sum)`).

Псевдокод алгоритма II для случая $P = m = n$ приведен в примере 5.11. Как и ранее, предполагается, что каждый процесс группы знает свой номер `myID` в группе и общее количество процессов в группе – n . Каналы `values_x[n]` напрямую не используются, поскольку отсутствуют операции двухточечного обмена, однако их следует объявить, поскольку они неявно задействованы в коллективных взаимодействиях. Глобальная функция связи `Scatter()`, как и в примере 5.10, распределяет исходную матрицу A (но уже по столбцам) и вектор \mathbf{x} по рабочим процессам. Глобальная операция приведения `Reduce()` используется для сборки результирующего вектора \mathbf{c} с суммированием, поскольку частичные суммы элементов \mathbf{c} распределены по процессам. Если использовать операцию `Allreduce()`, то результирующий вектор будет собран на каждом процессе.

Сравнивая два способа умножения матрицы на вектор, следует отметить следующее. При первом способе требуется либо организация распространения по кольцу частей вектора \mathbf{x} , либо его хранение целиком. При реализации второго способа в процессе хранится не только столбец матрицы A , но и весь вектор результата (хотя в свою локальную версию каждый процесс насчитывает только часть значения). Кроме того, в этом алгоритме необходимо выполнить дорогую коллективную операцию `Reduce()`. Заметим, что в численных методах задача вычисления произведения матрицы

на вектор не является самостоятельной, а выбор способа распараллеливания зависит в основном от сложившейся ситуации распределения матрицы и вектора по процессам и в памяти.

Пример 5.11. Умножение матрицы на вектор. Псевдокод алгоритма П

```
chan values_x[n](int);
thread MatrixVectorProduct2 (myID=0; myID<n){
  if (myID == 0) {
    double aa[n,n], xx[n]; //исходные матрица A и столбец x
    double cc[n];          // результирующий вектор c
    инициализировать a и x;
  }
  double a[n], x; //столбец myID матрицы A и элемент вектора x
  double c[n]=0; //вектор частичных сумм для результата C
  int i
  //Получить столбец myID матрицы A от корневого процесса
  Scatter(aa[*][], root=0, a[*]);
  //Получить элемент myID вектора x от корневого процесса
  Scatter(xx[], root=0, x);
  //Вычислить свою часть элементов вектора c
  for (i=0; i<n; i++) c[i]=a[i]*x;
  //В корневом процессе собрать вектор c как поэлементную
  сумму
  Reduce(c[], root=0, cc[], sum);
  if (i == 0) вывести результат, который теперь в векторе cc;
}
```

Умножение матриц

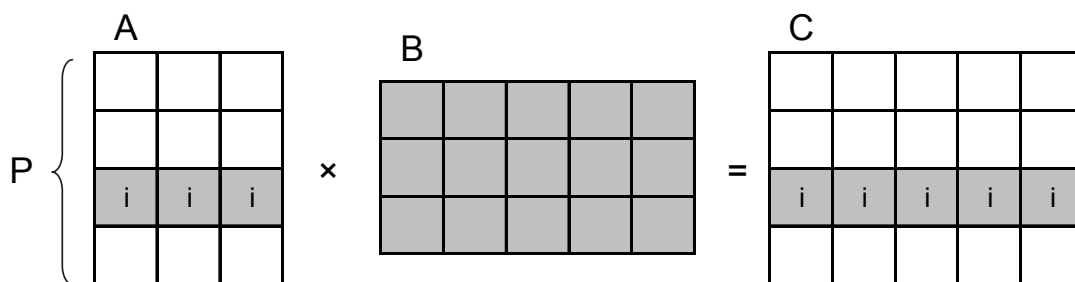
Задача 5.4. Пусть даны матрицы A и B размерностью $m \times k$ и $k \times n$ соответственно. Необходимо найти произведение матриц AB размерности $m \times n$.

При обсуждении основных парадигм параллельного программирования в гл. 1 было рассмотрено несколько параллельных реализаций алгоритма умножения матрицы на матрицу (примеры 1.1–1.5, 1.7). Сам алгоритм вычисления произведения матриц имеет хороший внутренний параллелизм. Основные отличия заключаются в подходе к распределению вычислительной нагрузки, т. е. декомпозиции матриц по процессам.

При реализации алгоритма для BC с общей памятью можно выделить два основных подхода: 1) статическая декомпозиция по блокам строк и столбцов (примеры 4.4, 4.6); 2) динамическая декомпозиция в рамках парадигмы «портфель задач» (пример 4.17).

При реализации алгоритма для BC с распределенной памятью обычно ограничиваются статическим распределением данных по процессам и рассматривают три основных способа декомпозиции, которые отличаются топологией организации связей [25, 33, 35].

Алгоритм I (топология «кольцо»). Матрица A разрезается на полосы по строкам, а матрица B – на полосы по столбцам. Количество процессов соответствует количеству полос матрицы A . Процессы связаны в логическую топологию «кольцо», по которому циркулируют полосы матрицы B . После полного круга каждый процесс будет иметь горизонтальную полосу результирующей матрицы (рис. 5.15). При необходимости матрицу можно собрать на каком-либо процессе.



5.15. Произведение матриц. Алгоритм I

Пример 5.10 содержит псевдокод, реализующий алгоритм для случая $m = n = k = P$, где P – количество процессов. Как и ранее, предполагается, что все процессы принадлежат одной группе, каждый процесс группы знает свой номер в группе и общее количество процессов в ней. Особенности нотации коллективных взаимодействий описаны в предыдущем пункте. Работа алгоритма для восьми процессов проиллюстрирована на рис. 1.14.

Легко вычислить количество двухточечных обменов, необходимых для реализации алгоритма на P процессах: $2P$ (подготовка данных) + $(P-1)P$ (вычисления) + $(P-1)$ (сборка результата).

Алгоритм II (топология «2D-решетка»). Матрица A разрезана на полосы по строкам, а матрица B – на полосы по столбцам. В отличие от алгоритма I параллельно вычисляются блоки элементов результирующей матрицы C .

Пусть P процессов образуют виртуальную топологию «2D-решетка» с размерностями P_1 и P_2 ($P = P_1 \times P_2$) в каждом из направлений. В такой топологии процессу, наряду с его линейным номером $rank$, $rank = 0, \dots, P-1$, можно сопоставить узел двумерной решетки с декартовыми координатами (i, j) , $i = 0, \dots, P_1-1, j = 0, \dots, P_2-1$.

Вдоль граней решетки перед расчетами распространяются строки матрицы A и столбцы матрицы B . После цикла вычислений каждый процесс будет иметь блок элементов результирующей матрицы C (рис. 5.16). При необходимости матрицу C можно собрать в памяти какого-либо процесса.

Пример 5.12. Произведение матриц. Псевдокод алгоритма I – топология «кольцо»

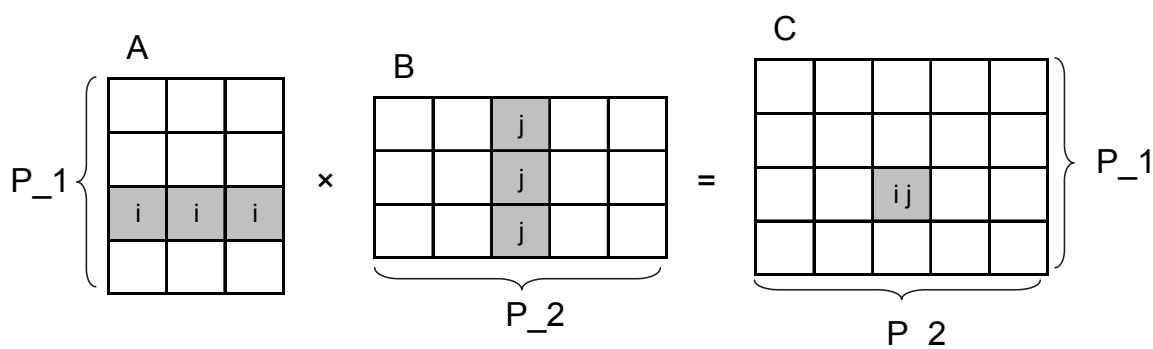
```

chan values_x[n](int b[n]);
thread MatrixProduct1 (i=0; i<n){
  if (i == 0) {
    double aa[n][n], bb[n][n]; //исходные матрицы A и B
    double cc[n][n]; //результатирующая матрица C
    инициализировать aa и bb;
  }
  double a[n],b[n],c[n]; //строка A, столбцы B и C
  double sum = 0.0; //для промежуточных произведений
  int next_p, prev_p; //номера соседних процессов
  //Определение по своему номеру i номеров соседних процессов
  if (i == n-1) next_p = 0; else next_p = i+1;
  if (i == 0) prev_p = n-1; else prev_p = i-1;
  //Получить строку i матрицы A от корневого процесса
  Scatter(aa[][*], root=0, a[*]);
  //Получить столбец i матрицы B от корневого процесса
  Scatter(bb[*][i], root=0, b[*]);
  // пустить по кругу столбцы b[] и вычислить cc[i, *]
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum = sum + a[k]*b[k];
    l=(i+n-j)%n
    c[l] = sum;
    //Отправить мой столбец матрицы b следующему процессу
    send values_x[next_p](b);
    //Получить новый столбец матрицы b от предыдущего процесса
    receive values_x[i](b);
  }
  //Вычислить оставшийся элемент
  //cc[i,i+1] = aa[i,*] * bb[* ,i+1]
  for (k=0; k<n-1; k++)
    sum = sum + a[k]*b[k];
  c[i+1] = sum;
  //Собрать результат на корневом процессе
  Gather(c[], root=0, cc[][*]);
  if (i == 0) вывести результат, который теперь в матрице cc;
}

```

Пример 5.13 содержит основной псевдокод алгоритма вычисления произведения квадратных матриц размерности $n \times n$ на декартовой 2D-решетке размерностью $P_1 \times P_2$, $P_1 = n$, $P_2 = n$. Предполагается, что каждый процесс группы знает свой линейный номер $myID$ в группе и общее количество процессов в группе $P_1 \times P_2 = n \times n$. Кроме того, с каждым процессом сопоставлены декартовы координаты (myX , myY). Все данные вначале сосредоточены на нулевом процессе.

В используемой нотации для групповых функций первый аргумент соответствует буферу, из которого производится рассылка, второй аргумент определяет корневой процесс, третий – буфер приема в принимающем процессе. Если коллективная рассылка выполняется на декартовой решетке, то появляется четвертый аргумент – декартово направление, вдоль которого выполняется коллективная операция. Символ «*» в размерности массива в глобальных операциях связи означает, что по этому направлению *не* проводится декомпозиция или сборка данных, символ «#» в четвертом аргументе указывает декартово направление, вдоль которого выполняется операция. Например, функция `Scatter(aa[][*], root=(0,0), a[], (#,0))`; распределяет двумерный массив `aa`, изначально хранящийся в процессе `(0,0)`, по строкам по всем процессам с декартовыми координатами `(myX, 0)`, `myX=0, 1, ..., P_1-1`; результат операции сохраняется в массиве `a`. Операция `Broadcast(...)` без изменения рассылает данные, на которые указывает первый аргумент, от корневого процесса с декартовым номером, указанным во втором аргументе, вдоль направления, обозначенного четвертым аргументом, сохраняя полученные данные в буфер, указанный в третьем аргументе. Например, `Broadcast(a[], root=(myX,0), a[], (myX,#))`; без изменения рассылает строку `a[]` от корневого процесса с декартовым номером `(myX, 0)` вдоль направления `y`, сохраняя в буфер `a[]`.



5.16. Произведение матриц. Алгоритм II

Легко вычислить количество двухточечных обменов, необходимых для реализации алгоритма $P_1 + P_1(P_2-1) + P_2 + P_2(P_1-1)$ (начальная рассылка данных) + (P_1P_2-1) (сборка результата). Следует отметить, что при вычислениях в этом алгоритме достигается максимальная параллельность, а количество обменов при рассылке и сборе информации то же, что и в алгоритме I. В вычислительных системах с топологией связей 2D-тор передачи данных могут идти одновременно вдоль обоих направлений (каждый процессор связан с четырьмя соседями). Для этих топологий алгоритм II наиболее эффективен.

Пример 5.13. Фрагмент алгоритма вычисления произведения матриц в топологии «2D-решетка»

```

thread MatrixProduct2 (myID =0; myID <P_1*P_2) {
  if (i == 0) {
    double aa[n][n], bb[n][n]; //исходные матрицы A и B
    double cc[n][n]; //результатирующая матрица C
    инициализировать aa и bb;
  }
  //Отображение линейного номера на координаты
  //декартовой 2D-решетки
  myID --> (myX, myY);
  double a[n], b[n], c[n]; //строка A, столбцы B и C
  double sum = 0.0; //для промежуточных произведений
  ...
  //Распределение данных матрицы A по строкам
  //с 0-го процесса вдоль направления x
  Scatter(aa[myX][*], root=(0,0), a[], (#,0));
  //Рассылка строк матрицы A
  //с процесса с координатами (myX, 0) вдоль направления y
  Broadcast(a[], root=(myX,0), a[], (myX,#));
  //Распределение столбцов матрицы B
  //с 0-го процесса вдоль направления y
  Scatter(bb[*][myY], root=(0,0), b[], (0,#));
  //Рассылка столбцов матрицы B
  //с процесса с координатами (0, myY) вдоль направления x
  Broadcast(b[], root=(0,myY), b[], (#,myY));
  //Вычисление элемента матрицы cc
  //cc[myX][myY] = cc[myX][myY] + aa[myX][k] * bb[k][myY]
  c = 0;
  for (k=0 ; k<n; k++)
    c += a[k] * b[k];
  //Сборка результата на 0-м процессе
  Gather(c, root=0, c[][]);
  ...
}

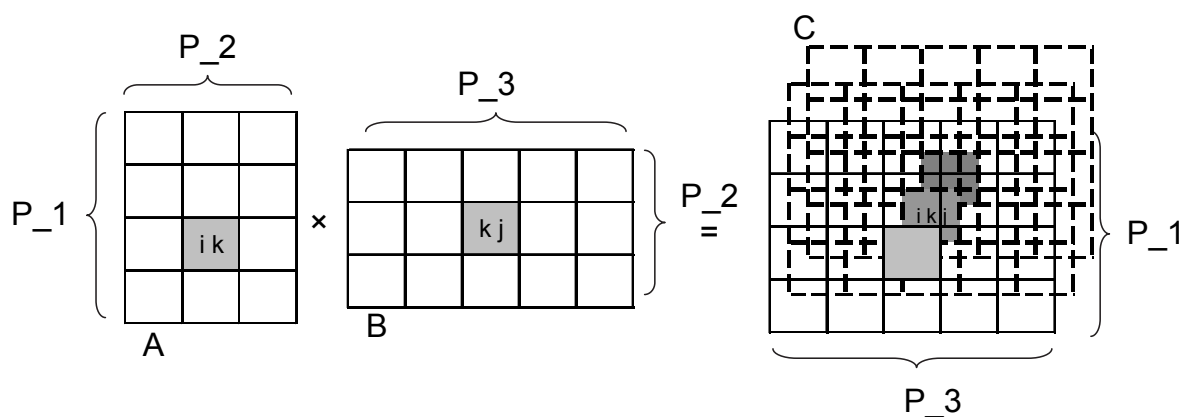
```

Алгоритм III (топология «3D-решетка»). Если провести декомпозицию матриц A и B по блокам строк и столбцов одновременно, то каждый процесс будет вычислять произведение блоков $A_{ik} \times B_{kj}$, которые являются частичными суммами для элемента c_{ij} блока результирующей матрицы C_{ij} .

Пусть P процессов образуют виртуальную топологию «3D-решетка» с размерностями P_1 , P_2 и P_3 ($P = P_1 \times P_2 \times P_3$) в каждом из направлений. В такой топологии процессу, наряду с его линейным номером $rank$, $rank = 0, \dots, P-1$, можно сопоставить узел трехмерной решетки с декартовыми координатами (i, k, j) , $i = 0, \dots, P_1-1$, $k = 0, \dots, P_2-1$, $j = 0, \dots, P_3-1$.

Вдоль граней 3D-решетки перед расчетами распространяются блоки матриц A и B . Процесс с декартовыми координатами (i, k, j) вычисляет произведение блоков $A_{ik} \times B_{kj}$, элементы которого являются только частичными суммами для соответствующих элементов в блоке результирующей матрицы C_{ij} (рис. 5.17). Полные элементы результирующей матрицы собираются на грани $(i, 0, j)$ с помощью операции редукции с суммированием вдоль второго направления ($\text{Reduce}(\dots \text{Sum} \dots)$). При необходимости результирующую матрицу можно собрать в каком-либо процессе.

Пример 5.14 содержит основной псевдокод алгоритма вычисления произведения матриц размерности $n \times n$ на декартовой 3D-решетке размерностью $P_1 \times P_2 \times P_3$, $P_1 = n$, $P_2 = n$, $P_3 = n$. Предполагается, что каждый процесс группы знает свой линейный номер myID в группе и общее количество процессов в группе $P_1 \times P_2 \times P_3 = n \times n \times n$. Кроме того, с каждым процессом сопоставлены декартовы координаты $(\text{myX}, \text{myY}, \text{myZ})$. Все данные вначале сосредоточены на нулевом процессе.



5.17. Произведение матриц. Алгоритм III

Нотации глобальных функций связи $\text{Scatter}()$, $\text{Gather}()$ и $\text{Broadcast}()$ аналогичны соответствующим нотациям примеров 5.12 и 5.13 с учетом возможности определения 3D-декартовых координат процессов. Например, функция $\text{Scatter}(\text{aa}[\][\], \text{root}=(0,0,0), a, (\#, \#, 0))$ распределяет матрицу A , изначально хранящуюся в процессе $(0,0,0)$, по элементам по всем процессам с декартовыми координатами $(\text{myX}, \text{myY}, 0)$, $\text{myX}=0, 1, \dots, P_1-1$, $\text{myY}=0, 1, \dots, P_2-1$. Функция приведения $\text{Reduce}(\dots)$ используется для нахождения блоков элементов $c[i, j]$ результирующей матрицы C , частичные суммы $c[i, k, j]$ этих элементов суммируются вдоль направления, указанного в четвертом параметре, и в результате оказываются распределены по процессам на грани $(i, 0, j)$. В целом нотация функции $\text{Reduce}()$ такая же, как в примере 5.8.

Легко вычислить количество двухточечных обменов, необходимых для реализации алгоритма $(P_1P_2 - 1) + P_1P_2(P_3 - 1) + (P_2P_3 - 1) + P_2P_3(P_1 - 1)$ (начальная рассылка данных) + P_1P_2 (вычисления) + P_3P_1 (сборка результата). В ВС с топологией связей 3D-тор передачи данных могут идти одновременно вдоль трех направлений (каждый процессор связан с шестью соседями). Для этих топологий алгоритм III наиболее эффективен.

Пример 5.14. Фрагмент псевдокода вычисления произведения матриц в топологии «3D-решетка»

```
thread MatrixProduct2 (myID=0; myID <P_1*P_2*P_3){
  if (myID == 0) {
    double aa[n][n], bb[n][n]; //исходные матрицы A и B
    double cc[n][n]; //результатирующая матрица C
    инициализировать aa и bb;
  }
  //Отображение линейного номера на координаты
  //декартовой 3D-решетки
  myID --> (myX,myY,myZ);
  double a,b,c; // элементы (myX,myY) матриц A, B, C
  ...
  //Распределение данных матрицы A по элементам
  //с 0-го процесса вдоль направлений x и y
  Scatter(aa[0][0], root=(0,0), a, (#,#,0));
  //Распространение данных матрицы A с каждого процесса
  //с координатами (i,k,0) вдоль направления z
  Broadcast(a, root=(i,j,0), a, (i,k,#));
  //Распределение данных матрицы B по элементам
  //с 0-го процесса вдоль направлений y и z
  Scatter(bb[0][0], root=(0,0,0), a, (0,#,#));

  //Распространение данных матрицы B с каждого процесса
  //с координатами (0,k,j) вдоль направления x
  Broadcast(b, root=(0,j,k), b, (#,k,j));
  //Вычисление частичных сумм  $C(i,k,j) = A(i,k) \times B(k,j)$ ;
  c=a*b
  //Сборка вдоль направления y на процессах (i,0,k)
  //элемента c(i,k) результирующей матрицы
  Reduce(c, root=(i,0,k), c, (i,#,k), sum);
  //Сборка результата на 0-м процессоре
  Gather(c, root=(0,0,0), c[0][0]);
  ...
}
```

Решение СЛАУ методом Гаусса

Задача 5.5. Найти вектор \mathbf{x} , являющийся решением системы линейных алгебраических уравнений (СЛАУ)

$$A\mathbf{x} = \mathbf{f}.$$

Рассмотрим возможные параллельные варианты прямого метода решения этой системы – метода Гаусса [25, 33, 35]. Метод Гаусса состоит из 1) прямого хода – последовательного исключения неизвестных приведением матрицы к верхнетреугольному виду, 2) обратного хода – получения вектора решений.

Алгоритм I предполагает блочный метод распределения данных – матрица и вектор правых частей разрезаются на полосы, которые распределяются по процессам. Вектор результатов также оказывается разрезан на полосы и распределен. При таком распределении данных прямой и обратный ходы фактически происходят последовательно по полосам.

Схематично этот алгоритм кратко можно записать следующим образом.

1. Матрица A и вектор правых частей \mathbf{f} разрезаются на горизонтальные полосы. Каждый процесс должен хранить полосу матрицы A и вектора \mathbf{f} . Полосы распределяются по процессам – `Scatter(...)`.

2. Прямой ход проводится в процессах последовательно начиная с нулевого. К треугольному виду последовательно приводятся строки каждого процесса с передачей каждой вычисленной строки всем процессам, номера которых превышают текущий – `Broadcast(...)`. Заметим, что деление строк на коэффициенты при x_i не требует информации от других процессов.

3. Обратный ход проводится также последовательно каждым процессом, начиная с последнего. Информацию в обратном ходе лучше передавать обо всей найденной полосе сразу – `Broadcast(...)`.

4. Результат окажется собранным на нулевом процессе.

Недостатки этого алгоритма очевидны – и прямой, и обратный ходы фактически выполняются последовательно, большинство процессов простаивают. Повышения эффективности можно ожидать на пути более равномерного распределения вычислительной нагрузки по процессам. Для этого достаточно изменить способ распределения матрицы A и вектора правых частей \mathbf{f} по процессам.

Алгоритм II предполагает циклическую блочную схему распределения данных – матрица и вектор правых частей разрезаны на строки и циклически распределены по узлам. Вектор результатов также распределен поэлементно по узлам. Прямой и обратный ходы фактически происходят веретеном построчно.

Пусть для простоты n кратно P , где P – количество процессов. Кратко алгоритм можно записать следующим образом.

1. Матрица A и вектор правых частей \mathbf{f} разрезаются на строки. Каждый процесс должен хранить несколько строк матрицы A и соответствующих элементов вектора \mathbf{F} (n/P раз `Scatter(...)` полосы из P строк).

2. Прямой ход проводится в порядке обычной линейной нумерации строк, начиная с нулевого процесса. К треугольному виду приводится циклически по одной строке в каждом процессе с передачей вычисленной строки всем узлам (n раз `Broadcast(...)`).

3. Обратный ход проводится также циклически в порядке обычной линейной нумерации строк, начиная с последнего процесса (n раз `Broadcast()`).

4. Результат окажется собранным на всех узлах.

Заметим, что в этом алгоритме процессоры загружены более равномерно, чем в первом, однако алгоритм менее выгоден по количеству обменов. Эффективность второго алгоритма проявляется на матрицах большой размерности, когда время счета на полосе становится значительным и последовательная их обработка является крайне невыгодной.

Решение СЛАУ методом простой итерации

Рассмотрим следующий итерационный метод решения СЛАУ:

$$\begin{aligned} x_i^{(0)} &= x_i^0, \\ x_i^{(k+1)} &= x_i^{(k)} - \tau \left(\sum_{j=1}^n a_{ij} x_j^{(k)} - f \right), \quad i = 1, \dots, n, \end{aligned} \quad (5.12)$$

где $\mathbf{x}^{(0)} = (x_1^0, \dots, x_n^0)^T$ – вектор начального приближения; τ – параметр метода.

Для реализации алгоритма на P процессах исходную матрицу коэффициентов A разрезаем на P горизонтальных полос; на соответствующие полосы разрезаются и векторы правой части \mathbf{f} , начального приближения $\mathbf{x}^{(0)}$ и следующего итерационного приближения $\mathbf{x}^{(k+1)}$. Полосы последовательно распределяются по соответствующим процессам, аналогично алгоритму 1 нахождения произведения матрицы на вектор.

Выражение $\sum_{j=1}^n a_{ij} x_j^{(k)}$ суть произведение матрицы и вектора, параллельная реализация которого обсуждалась выше. Заметим, что результат этого произведения не нужно полностью собирать на всех процессах, каждому процессу достаточно знать свою полосу произведения.

Считается, что решение найдено (итерационный метод сошелся), если: 1) либо некоторая норма разности двух последовательных приближений меньше заранее заданной точности ε :

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \varepsilon;$$

2) либо некоторая норма вектора невязки достаточно мала:

$$\|A\mathbf{x}^{(k+1)} - \mathbf{f}\| < \varepsilon.$$

Заметим, что невязка неявно вычисляется, поскольку элементы вектора невязки стоят в формуле (5.12) в скобках.

Наиболее распространенной нормой, рассматриваемой для остановки итерационного процесса, является равномерная норма:

$$\|\mathbf{x}\| = \max_{i=1, \dots, n} \{x_i\}.$$

Поскольку вектор невязки или векторы старого и нового приближения распределены по процессам, то необходимо периодически вычислять максимум по всем узлам $\text{Reduce}(\dots, \text{Max}, \dots)$. В идеале это необходимо делать на каждой итерации. Однако для сокращения количества обменов можно проводить операцию приведения каждые несколько итераций, найдя компромисс между временем счета одной итерации (зависит от размерности системы и количества вычислительных узлов) и временем операции приведения (зависит от количества узлов).

Таким образом, решение СЛАУ методом простой итерации реализует алгоритм пульсации (пример 5.6), в котором обмены с соседними процессами усложнены вычислением произведения матрицы на вектор, а барьер в конце итерации неявно выполняется на операции приведения $\text{Reduce}(\dots, \text{Max}, \dots)$.

Решение задачи Дирихле для уравнения Пуассона методом Якоби

Пусть $\Omega = (0,1) \times (0,1)$ – единичный квадрат с границей Γ . Рассмотрим краевую задачу Дирихле для двумерного уравнения Пуассона в следующем виде:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \quad \text{в } \Omega, \quad (5.13)$$

$$u = u_0|_{\Gamma}. \quad (5.14)$$

Для численного решения задачи рассмотрим без уменьшения общности равномерную квадратную сетку с шагом h :

$$\bar{\omega}^h = \{(x_i, y_j) : x_i = ih, y_j = jh, i, j = 0, \dots, n\}.$$

Обозначим через ω^h множество ее внутренних узлов. Конечно-разностная аппроксимация даст явную разностную схему следующего вида:

$$u_{i,j} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2 f_{i,j}), \quad i, j = 1, \dots, n-1, \quad (5.15)$$

$$u_{i,j}|_{\Gamma} = u_0(x_i, y_j), \quad (5.16)$$

где $u_{i,j} = u(x_i, y_j)$. Для решения системы (5.15)–(5.16) воспользуемся итерационным методом Якоби в следующем виде:

$$u_{i,j}^{(k+1)} = \frac{1}{4} \left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - h^2 f_{i,j} \right), \quad i, j = 1, \dots, n-1, \quad k = 1, 2, \dots, \quad (5.17)$$

$$u_{i,j}^{(k)}|_{\Gamma} = u_0(x_i, y_j), \quad \forall k.$$

Код основной части последовательной программы нахождения решения представлен в примере 5.15.

Пример 5.15. Последовательная реализация метода Якоби для решения задачи Дирихле для уравнения Пуассона

```
...
//Норма и точность для останова итерационного процесса
double norm=10., eps=1.E-5
//численное решение на двух соседних итерациях,
//вектор правой части
double u[n+1][n+1], u_new[n+1][n+1], f[n+1][n+1];
... //инициализация массивов,
    //включая граничные значения, которые не изменяются
while(norm<eps){ // основной цикл
    //Очередная итерация
    for(j = 1; j < n; j++)
        for(i = 1; i < n; i++)
            u_new[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                                u[i][j-1]+u[i][j+1] -h*h*f[i][j]);
    //Вычисление нормы разности между двумя соседними итерациями
    ...
    //Подготовка массива для очередной итерации
    for(j = 1; j < n; j++)
        for(i = 1; i < n; i++)
            u[i][j] = u_new[i][j];
}
... // вывод результатов
```

Поскольку цикл подготовки массивов для очередной итерации занимает достаточно много времени, но не производит полезной работы, от него можно избавиться. Если язык программирования позволяет, то можно использовать его средства для эффективного обмена значениями массивов. Например, в коде на Си/Си++ быстро скопировать массив `u_new` в массив `u` поможет функция

```
memcpy(u, u_new, n*n*sizeof(double));
```


Вместо обмена значениями можно поменять местами адреса массивов, воспользовавшись дополнительным указателем:

```
double **ptr;
ptr=u; u=u_new; u_new=ptr;
```

Часто удобно выполнить *развертку цикла* (пример 5.16). Суть развертки цикла заключается в том, что начальный код тела цикла повторяется в теле развернутого цикла несколько раз, уменьшая во столько же раз количество итераций развернутого цикла. Для циклов `for` при развертке следует увеличить шаг цикла.

Сама по себе развертка цикла мало влияет на производительность алгоритма. Она может даже его ухудшить, если команды тела развернутого цикла не умещаются в кэше команд процессора. Однако применение развертки цикла часто позволяет использовать другую оптимизацию алгоритма. В данном случае мы избавились от необходимости тратить время на холостую с точки зрения вычислений операцию обмена местами массивов `u` и `u_new`.

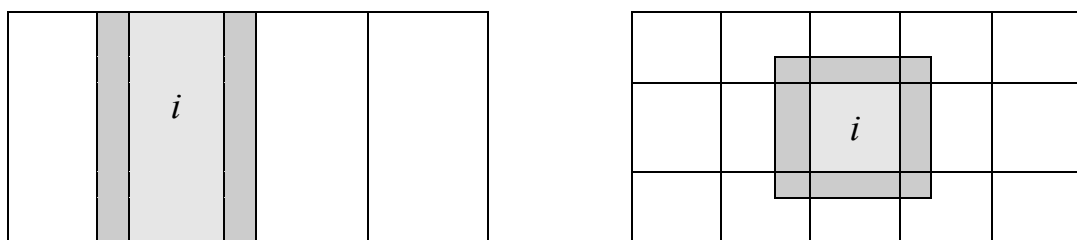
Пример 5.16. Развертка цикла

```
while(norm<eps){ // основной цикл
//Очередная итерация
for(j =1; j < n; j++)
  for(i = 1; i < n; i++)
    u_new[i][j]=0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]
                      -h*h*f[i][j]);
//Еще очередная итерация
for(j =1; j < n; j++)
  for(i = 1; i < n; i++)
    u[i][j]=0.25*(u_new[i-1][j]+u_new[i+1][j]
                  +u_new[i][j-1]+u_new[i][j+1]-h*h*f[i][j]);
//Вычисление нормы разности между двумя соседними итерациями
...
}
```

При распараллеливании алгоритма необходимо узлы сетки (а значит, и все массивы) распределить между процессами. Вопросы, связанные с эффективной декомпозицией прямоугольной области с регулярной сеткой, обсуждаются в гл. 6. Здесь отметим, что для данной задачи удачной будет декомпозиция области на вертикальные полосы или прямоугольные блоки (рис. 5.18).

При вычислении значения функции u в узле с номером (i, j) участвуют узлы с $i-1, i+1, j-1, j+1$ слоями. Для корректного вычисления в узлах

сетки, попадающих при декомпозиции на границы полос или блоков, необходимо знать текущие значения массивов u , u_{new} и f в узлах, принадлежащих соседним процессам. Следовательно, в каждом процессе необходимо хранить не только данные, вычисляемые процессом, но и часть данных, вычисляемых соседними процессами. Например, при методе Якоби размерности массивов u и u_{new} в каждом процессе должны быть на 2 больше в обоих направлениях, чем размерность данных, вычисляемых в этом процессе. В начале программы каждый процесс должен узнать, какие процессы являются для него соседними, можно создать специальные коммуникаторы для обмена сообщениями с соседями. После каждого цикла, помеченного комментарием «очередная итерация», следует производить обмен граничными значениями с процессами-соседями. Для исключения проблемы гонок каждая итерация требует барьерной синхронизации, которая часто неявно выполняется в функциях обмена.



5.18. Способы распределения данных по узлам. «Теневые» грани

Для оптимизации обменов можно также совмещать вычисления и передачу данных. Для этого в каждом процессе вначале вычисляются граничные элементы массивов, после чего инициируется передача этих элементов соседним процессам, а затем продолжается вычисление внутренних точек массивов. В результате передача данных и вычисления будут частично перекрываться во времени.

Если можно оценить количество итераций, требуемых для достижения приемлемой точности, то следует отказаться от оценки ошибки, требующей глобальной операции приведения, и заменить цикл `while` на цикл `for` с фиксированным количеством итераций.

5.4. Проблемы выбора эффективной параллельной реализации

Метод конечных элементов для задачи Дирихле для уравнения Пуассона

Как правило, даже при выбранном численном методе решения практической вычислительной задачи и известной архитектуре вычислительной

системы, для которой пишется параллельная программа, остается некоторый произвол в выборе технологии параллельного программирования, а также есть несколько вариантов непосредственной программной реализации численного метода (вспомните, сколько вариантов распараллеливания мы обсудили только для алгоритма вычисления произведения двух матриц!). Поскольку написание и отладка параллельной программы являются очень трудо- и времезатратными операциями, то важными становятся не только предварительная оценка внутреннего параллелизма численного метода, но и учет следующих факторов, влияющих в конечном итоге на эффективность распараллеливания:

- простота параллельной реализации численного метода в рамках выбранной технологии параллельного программирования;
- возможность изменения реализации численного алгоритма, связанная с уменьшением дополнительного времени T_p^{over} в оценке (5.2) общего времени выполнения параллельной программы T_p ;
- возможность изменения реализации численного алгоритма, связанная с оптимизацией доступа в память для уменьшения составляющей T_p^{mem} при оценке (5.2) общего времени выполнения параллельной программы T_p .

Обсудим решение этих проблем на примере выбора параллельной реализации решения задачи Дирехле для уравнения Пуассона с помощью метода Галеркина.

Использование метода конечных элементов (МКЭ) при пространственной дискретизации многих моделей математической физики дает ряд преимуществ перед методом конечных разностей, основное из которых – возможность использовать неструктурированные, неравномерные сетки для вычислительных областей сложной формы. В то же время МКЭ, безусловно, имеет большую вычислительную сложность [19] по сравнению с методом конечных разностей, что делает актуальным использование высокопроизводительных вычисленных систем для его реализации [66, 69, 71, 74–76].

Проведем исследование эффективности нескольких параллельных реализаций метода конечных элементов для задачи Дирехле для уравнения Пуассона (5.13)–(5.14), выполненных с помощью технологий MPI, OpenMP и совмещения этих технологий для SMP-узлов кластера.

Теория метода конечных элементов подробно изложена, например, в работах [15, 19, 31, 38]. Здесь приведем сведения, необходимые только для понимания обсуждаемых далее фрагментов программы.

Без потери общности будем искать решение задачи Дирихле с нулевым граничным условием (в (5.14) $u_0|_{\Gamma} \equiv 0$). Для построения метода Галер-

кина сформулируем задачу в слабом смысле. Умножим уравнение (5.13) на произвольную функцию $v \in \overset{\circ}{W}_2^1(\Omega)$, т. е. функцию непрерывную, кусочно непрерывно дифференцируемую и равную нулю на границе Γ . Проинтегрируем полученное равенство по области Ω . После применения интегрирования по частям получим тождество

$$\int_{\Omega} \left(\frac{\partial u}{\partial x} \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \frac{\partial v}{\partial y} \right) d\Omega = \int_{\Omega} f v d\Omega, \quad (5.18)$$

верное для $f \in L_2(\Omega)$ и $\forall v \in \overset{\circ}{W}_2^1(\Omega)$. Левая часть тождества (5.18) суть билинейная форма, а его правая часть – линейная форма. Обозначим их через $a(u, v)$ и $f(v)$ соответственно.

Перенумеруем некоторым линейным образом узлы сетки $\bar{\omega}^h$, например в лексикографическом порядке:

$$\bar{\omega}^h = \{p_k : p_k = (x_i, y_j), i, j = 0, \dots, n; k = 1, \dots, (n+1)^2\}.$$

Отметим, что пара (i, j) задает глобальную нумерацию узлов. Обозначим через γ^h множество граничных узлов:

$$\gamma^h = \{p_k : p_k = (x_i, y_j), i = 0, n, j = 0, \dots, n; i = 0, \dots, n, j = 0, n; k = 1, \dots, 4n\}.$$

На сетке $\bar{\omega}^h$ введем триангуляцию \mathfrak{T} , разделив каждую ячейку сетки на два треугольника диагональю, параллельной биссектрисе первого координатного угла. Перенумеруем полученные треугольные элементы $\mathfrak{T} = \{e_k\}_{k=1}^{2n^2}$. Рассмотрим набор базисных функций $\{\varphi_k\}_{k=1}^{(n-1)^2}$ («крышек» Куранта); непрерывных на Ω , равных нулю на Γ , линейных на каждом элементе e_i и удовлетворяющих условию

$$\varphi_k(p_l) = \begin{cases} 1, & p_l = p_k, p_l \in \omega^h; \\ 0, & p_l \neq p_k. \end{cases}$$

На рис. 5.19 изображен носитель функции φ_k . Обозначим через

$$H^h = \text{span}\{\varphi_k\}_{k=1}^{(n-1)^2}$$

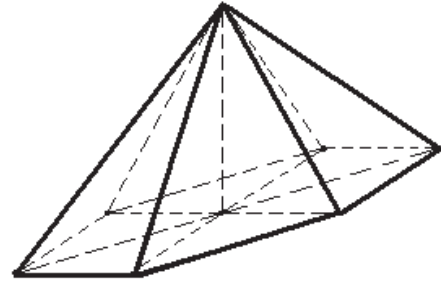


Рис. 5.19. Вид функции φ_k

линейную оболочку базисных функций. Тогда метод Галёркина можно сформулировать в следующем виде: *найти функцию*

$$u^h = \sum_{k=1}^{(n-1)^2} u_k^h \varphi_k \in H^h, \quad (5.19)$$

где $u_k^h = u^h(p_k)$, удовлетворяющую тождеству

$$a(u^h, v^h) = f(v^h) \quad \forall v^h \in H^h. \quad (5.20)$$

Поскольку тождество (5.19) верно для любой функции $v^h \in H^h$, то в качестве v^h достаточно рассмотреть базис в пространстве H^h . Тогда (5.20) соответствует системе уравнений

$$a(u^h, \varphi_k) = f(\varphi_k), \quad k = 1, \dots, (n-1)^2. \quad (5.21)$$

Систему (5.20) можно записать в матрично-векторном виде:

$$A_h \mathbf{u}^h = \mathbf{f}^h. \quad (5.22)$$

Здесь вектор \mathbf{u}^h неизвестных состоит из коэффициентов u_{ij}^h представления u^h (5.19), $\mathbf{f}^h = \{f_k^h = f(\varphi_k)\}_{k=1}^{(n-1)^2}$, а коэффициенты глобальной матрицы жесткости A_h можно рассчитать по следующим формулам:

$$a_{kl} = \int_{\Omega} \left(\frac{\partial \varphi_k}{\partial x} \frac{\partial \varphi_l}{\partial x} + \frac{\partial \varphi_k}{\partial y} \frac{\partial \varphi_l}{\partial y} \right) d\Omega, \quad k, l = 1, \dots, (n-1)^2. \quad (5.23)$$

Размерность всех векторов равна $(n-1)^2$, соответственно размерность матрицы A_h — $(n-1)^4$. С учетом возможности явного представления функций φ_k и их производных на каждом треугольном элементе носителя $\text{supp}(\varphi_k)$, а также вычисления интегралов (5.23) по всему носителю (в более сложных случаях применяют подходящие квадратурные формулы) решение задачи (5.13)–(5.14) сводится к решению системы линейных алгебраических уравнений (5.23).

Отметим, что для равномерной согласованной триангуляции прямоугольной области структура A_h регулярная, а потому ее удобно хранить. Однако в общем случае для областей сложной формы и различных триангуляций структура A_h может быть сильно нерегулярной. Поэтому при программировании метода конечных элементов удобно не хранить глобальную матрицу жесткости (тем более, что она при правильном выборе базиса является сильно разреженной), а при необходимости собирать ее из локальных матриц жесткости $A_h^{\text{loc}}(e_k)$ элементов $e_k \in \mathfrak{T}$.

На рис. 5.20 изображен конечный элемент $e_k \in \mathfrak{T}$ с локальной нумерацией вершин (вершина 1 совпадает с прямым углом). На рис. 5.21 приведены элементы a_{kl}^{loc} локальной матрицы жесткости, которые вычисляются по следующим формулам:

$$a_{kl}^{\text{loc}} = \int_{e_k} \left(\frac{\partial \varphi_k}{\partial x} \frac{\partial \varphi_l}{\partial x} + \frac{\partial \varphi_k}{\partial y} \frac{\partial \varphi_l}{\partial y} \right) d\Omega, \quad k, l = 1, \dots, (n-1)^2.$$

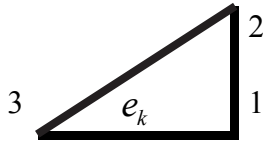


Рис. 5.20. Нумерация узлов в конечном элементе e_k

A_h^{loc} $l \backslash k$			
	1	2	3
1	1	-0,5	-0,5
2	-0,5	0,5	0
3	-0,5	0	0,5

5.21. Локальная матрица жесткости для элемента e_k

Применение различных методов решения СЛАУ, в том числе итерационных и многосеточных, как правило, требует не работы с матрицей A_h , а вычисления и хранения вектора невязки $\mathbf{r}^h = A_h \mathbf{u}^h - \mathbf{f}^h$, который тоже собирают на основании локальных матриц жесткости элементов.

Операция вычисления невязки может быть записана следующим образом. Для каждого элемента $e_k \in \mathfrak{T}$ вычисляется локальный вектор невязки $\mathbf{r}_{\text{loc}}^h = A_h^{\text{loc}} \mathbf{u}_{\text{loc}}^h - \mathbf{f}_{\text{loc}}^h$, где элементы векторов $\mathbf{r}_{\text{loc}}^h$, $\mathbf{u}_{\text{loc}}^h$, $\mathbf{f}_{\text{loc}}^h$ вычислены в вершинах треугольного конечного элемента e_k . Элементы $\mathbf{r}_{\text{loc}}^h$ являются частичными суммами элементов вектора невязки \mathbf{r}^h , причем при его вычислении номер узла следует брать в глобальной нумерации. В этом смысле невязка есть сумма локальных невязок:

$$\mathbf{r}^h = \sum_{e_k \in \mathfrak{T}} \mathbf{r}_{\text{loc}}^h. \quad (5.24)$$

Последовательная реализация различных стратегий сборки невязки

Самой вычислительно трудоемкой операцией в МКЭ является сборка невязки на основе локальных матриц жесткости элементов. Существует, по крайней мере, два способа (рис. 5.22) обхода триангуляции при сборке глобальной невязки: 1) по конечным элементам; 2) по узлам сетки.

Для каждого из способов требуется хранение в памяти специальной структуры данных, по-разному описывающих представление триангуля-

ции. Различным является и порядок сборки невязки. Каждая из этих особенностей влияет на производительность последовательной программы, возможности ее оптимизации и потенциальной эффективности параллельной версии. Рассмотрим каждый способ подробнее.

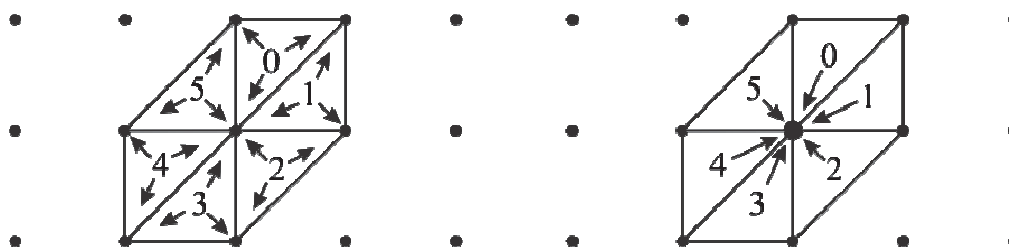


Рис. 5.22. Схема сборки невязки по элементам (слева) и по точкам сетки (справа)

Сборка невязки по элементам. Этот способ является традиционным для последовательных реализаций сборки, поскольку обеспечивает наиболее выгодное распределение памяти при хранении информации о триангуляции.

Как правило, триангуляция хранится в двух структурах.

1. *Описание узлов сетки.* Для каждого узла p_k хранятся его координаты x_k и y_k , а также, возможно, признак принадлежности узла границе (например, 0 – внутренний узел, 1 – граничный). Кроме того, могут понадобиться значения в каждом узле каких-либо параметров задачи, например глубины, ветрового напряжения, теплопроводности и т. д. Для описания узлов можно создать массив структур или хранить несколько отдельных массивов значений. В любом случае номер элемента массива будет соответствовать номеру узла в линейной глобальной нумерации.

2. *Описание триангуляции,* т. е. вхождения узлов в конечные элементы (в нашем случае – в треугольники). Для каждого элемента e_k необходимо хранить номера узлов-вершин в глобальной линейной нумерации. Например, в нашем случае вершина с номером 1 соответствует прямому углу, далее нумерация вершин идет по порядку против часовой стрелки. Таким образом, для каждого элемента e_k задано соответствие локальной и глобальной нумерации его вершин. Возможно, для каждого e_k необходимо также хранить признак принадлежности его граней (в нашем случае граней 1–2, 2–3, 1–3 треугольника) границе (например, 0 – внутренняя грань, 1 – граница), а также его площадь.

Код примера 5.17 содержит описание простейшего способа хранения информации о сетке и триангуляции, а также процедуру сборки невязки.

При сборке невязки внешний цикл обходит расчетную область по элементам, а внутренний – по вершинам текущего элемента. Таким образом, обход осуществляется по массиву $tr[i][j]$, хранящему для каждого i -го треугольника глобальные номера его вершин ($j = 1, 2, 3$).

Пример 5.17. Фрагмент кода сборки невязки по элементам

```
...
int i;
//Общее количество узлов сетки и треугольников
int num_pts, num_tr;
//Массивы размерностью num_pts+1:
//координаты узлов (в фрагменте не используются),
//численное решение на предыдущей итерации, невязка
double *x, *y, *u, *xi;
//Массив размерностью num_pts+1:
//Хранение данных о принадлежности узлов границе
int *ind_pts;
//Массив размерностью num_pts+1 на 3:
//Хранение данных о вхождении точек в треугольник
int **tr;
//Динамическое выделение памяти,
//считывание и подготовка данных,
...
//Сборка невязки в массив xi по компонентам  $\xi_{loc}^h$ 
//Цикл обхода области по треугольникам
for (i=1; i<=num_tr; i++){
    //Обход треугольника по узлам сетки:
    //Вершина 1 в локальной нумерации,
    //1-я строка локальной матрицы жесткости  $A_h^{loc}$ 
    //умножается на вектор  $u_{loc}^h$ 
    if (ind_pts[tr[i][1]])
        xi[tr[i][1]] = xi[tr[i][1]]+u[tr[i][1]]
                        -0.5*u[tr[i][2]]-0.5*u[tr[i][3]]
                        -h*h*f[tr[i][1]]; // (1)
    //Вершины 2 и 3 в локальной нумерации,
    //2-я и 3-я строки  $A_h^{loc}$  умножаются на вектор  $u_{loc}^h$ 
    for (j=2; j<=3; j++)
        if (ind_pts[tr[i][j]])
            xi[tr[i][j]] = xi[tr[i][j]]-0.5*u[tr[i][1]]
                            +0.5*u[tr[i][j]]-h*h*f[tr[i][j]]; // (1)
}
...
```

Заметим, что поскольку структура локальной матрицы жесткости A_h^{loc} отличается для узла с прямым углом и острыми, то соответствующие

формулы для накопления невязки в программе отличаются. Можно вывести формулы для расчета значений матрицы жесткости для произвольного треугольного конечного элемента, и тогда структура расчета в большинстве случаев будет однородной, но невязка будет зависеть от координат узлов.

Рис. 5.23 содержит пример равномерной согласованной триангуляции на сетке 5×5 точек. На рис. 5.23 представлен пример файла, описывающего структуру триангуляции на этой сетке.

Как уже отмечалось, использование сборки невязки по элементам выгодно с точки зрения хранения однородной структуры триангуляции, поскольку каждый конечный элемент содержит ровно три узла. Более того, применяя специальную нумерацию точек и элементов, можно добиться уменьшения количества обращений к оперативной памяти за счет эффективного использования кэш-памяти для массива `tr[i][j]`.

Глобальный номер							
элемента	вершины			элемента	вершины		
e_i	1	2	3	e_i	1	2	3
1	6	1	7	17	16	11	17
2	2	7	1	18	12	17	11
3	7	2	8	18	17	12	18
4	3	8	2	20	13	18	12
5	8	3	9	21	18	13	19
6	4	9	3	22	14	10	13
7	9	4	10	23	19	14	20
8	5	10	4	24	15	20	14
9	11	6	12	25	21	16	22
10	7	12	6	26	17	23	16
11	12	7	13	27	22	17	23
12	8	13	7	28	18	23	17
13	13	8	14	29	23	18	24
14	9	14	8	30	19	24	18
15	14	9	15	31	24	19	26
16	10	15	9	32	20	25	19

Рис. 5.23. Структура триангуляции на сетке 5×5 .

Сборка невязки по конечным элементам

Сборка невязки по узлам сетки. Этот способ сборки невязки требует размещения в памяти дополнительных в общем случае нерегулярных структур, отвечающих за хранение информации о триангуляции. Действительно, кроме описанных ранее структур, необходимо использовать информацию о том, в какое количество элементов e_k (в нашем случае треугольников) входит каждый узел сетки и каковы номера этих элементов (k) в глобальной нумерации (рис. 5.24).

Код примера 5.18 содержит описание простейшего способа хранения информации о сетке и триангуляции, а также процедуру сборки невязки.

Пример 5.18. Фрагмент кода сборки невязки по узлам сетки

```

...
int i;
//Общее количество узлов сетки и треугольников,
//количество треугольников, содержащих точку
int num_pts, num_tr, num_tr_pts;;
//Массивы размерностью num_pts+1:
//координаты узлов (в фрагменте не используются),
//численное решение на предыдущей итерации, невязка
double *x, *y, *u, *xi;
//Массив размерностью num_pts+1:
//Хранение данных о принадлежности узлов границе
int *ind_pts;
//Массив размерностью num_pts+1 на 3:
//Хранение данных о вхождении точек в треугольник
int **tr;
//Массив размерностью num_pts+1 на 7:
//Хранение структуры "узел-элементы"
int **struct_tr;
//Динамическое выделение памяти,
//считывание и подготовка данных,
...
//Сборка невязки в массив xi по компонентам  $\xi_{loc}^h$ 
//Цикл обхода области по узлам сетки
for (i=1; i<= num_pts; i++){
    if (ind_pts[i]) {
        //Количество треугольников, содержащих узел i
        num_tr_pts=str_tr[i][0];
        //Цикл обхода по элементам, в которые входит узел i
        for (j=1; j<= num_tr_pts; j++){
            //Определяем номер j-го элемента, содержащего узел i
            N_tr=str_tr[i][j];
            //Определяем локальный номер i-го узла в элементе и
            //вычисляем соответствующие компоненты невязки
            if (i == tr[N_tr][1])
                xi[i] = xi[i]+u[i]-0.5*u[tr[N_tr][2]]
                    -0.5*u[tr[N_tr][3]]-h*h*f[i];
            else xi[i] = xi[i]-0.5*u[tr[N_tr][1]]+0.5*u[i]-h*h*f[i];
        }
    }
}
...

```

В примере при сборке невязки внешний цикл по i обходит расчетную область по узлам сетки и использует нулевой элемент структуры $str_tr[i][0]$ для определения количества итераций внутреннего цикла ($str_tr[i][0]$ хранит количество элементов, в которые входит i -й узел). Внутренний цикл перебирает все треугольники с глобальными номерами

$\text{str_tr}[i][j]$, в которые входит i -й узел, и добавляет в невязку для этого узла вклад от каждого треугольника. Здесь i – номер узла в глобальной линейной нумерации, j – локальный номер треугольника в носителе базисной функции ϕ_i , $j=1, \dots, \text{str_tr}[i][0]$.

Глобальный номер узла p_k	Кол-во элементов, в которые входит p_k	Глобальные номера элементов, в которые входит p_k						
1	2	1	2	–	–	–	–	–
2	3	2	3	4	–	–	–	–
3	3	4	5	6	–	–	–	–
4	3	6	7	8	–	–	–	–
5	1	8	–	–	–	–	–	–
6	3	1	9	10	–	–	–	–
7	6	1	2	3	10	11	12	–
8	6	3	4	5	12	13	14	–
9	6	5	6	7	14	15	16	–
10	3	7	8	16	–	–	–	–
11	3	9	17	18	–	–	–	–
12	6	9	10	11	18	19	20	–
13	6	11	12	13	20	21	22	–
14	6	13	14	15	22	23	24	–
15	3	15	16	24	–	–	–	–
16	3	17	25	26	–	–	–	–
17	6	17	18	19	26	27	28	–
18	6	19	20	21	28	29	30	–
19	6	21	22	23	30	31	32	–
20	3	23	24	32	–	–	–	–
21	1	25	–	–	–	–	–	–
22	3	25	26	27	–	–	–	–
23	3	27	28	29	–	–	–	–
24	3	29	30	31	–	–	–	–
25	2	31	32	–	–	–	–	–

Рис. 5.24. Структура триангуляции на сетке 5×5 . Сборка невязки по узлам сетки

Как и в первом случае, структура локальной матрицы жесткости A_h^{loc} отличается для узла с прямым углом и острыми, при этом для алгоритма справедливы все предыдущие связанные с этим фактом замечания.

Поскольку количество элементов в носителях базисных функций варьируется, то структура $\text{str_tr}[i][j]$ неоднородная, а потому сделать эффективным доступ к памяти при таком способе сборки невязки невозможно.

Таким образом, *последовательная программа, реализующая в МКЭ сборку невязки по узлам, является менее эффективной, чем последовательная программа со сборкой невязки по конечным элементам.*

Численные эксперименты для областей сложной формы показали, что время выполнения последовательной программы при сборке невязки по элементам в 1,5 раза меньше, чем при сборке по узлам сетки, что можно объяснить более выгодным распределением памяти в первом случае.

Следует также отметить, что существуют мощные возможности оптимизации кода у современных компиляторов. Умелое использование ключей компиляции и других возможностей компилятора может существенно улучшить время выполнения последовательной программы. Особое внимание следует уделять возможностям компиляторов по управлению динамической памятью [17].

Архитектура вычислительных систем

Оценим эффективность параллельных версий МКЭ с двумя описанными стратегиями сборки невязки. Поскольку задача (5.21) является модельной и в вычислительном плане малотрудоемкой, то численное исследование ускорения и эффективности ее параллельных реализаций не дает возможности реально отследить все эффекты, связанные с распараллеливанием. В то же время эта задача сохраняет все проблемы, возникающие в более общих ситуациях. Поэтому для численных экспериментов была выбрана аналогичная, но вычислительно трудоемкая задача, для которой реализованы все варианты последовательных и параллельных программ. Результаты этих исследований приводятся в графиках, а рассуждения без потери общности будем выполнять на нашей модельной задаче.

Расчеты проводились на высокопроизводительных комплексах Сибирского федерального университета и Сибирского суперкомпьютерного центра СО РАН (ССКЦ СО РАН). В общем виде архитектура обоих вычислительных систем – это SMP-узловой кластер, т. е. каждый узел распределенного кластера является системой на общей памяти (рис. 5.25).

Ресурсы комплекса СФУ: 280 счетных узлов (каждый узел включает в себя 16 Gb оперативной памяти и два 4-ядерных процессора Xeon quad core E5345 2.33 GHz); дисковая подсистема IBM DS3400 объемом 20 Тб; подключаемые дисковые подсистемы IBM DS4700 до 200 Тб; высокоскоростная сеть InfiniBand 20 Гбит/с. Пиковая производительность комплекса СФУ согласно тесту Linpack достигает 16,872 Tflops.

На комплексе ССКЦ СО РАН вычисления проводились на SMP-узловом кластере, который включает 96 вычислительных узлов на 6-ядерных процессорах Intel Xeon X5670 2.93 GHz (Westmere) (всего 192 процессора и 1 152 ядра), блейд-серверы HP BL2x220c G7, 24 GB RAM (на узел), 96 Гбайт ОЗУ, 6 Гбайт памяти GDDR5; общая пиковая производительность – 13,5 Tflops.

Для исследования OpenMP-программ вычисления также проводились на сервере ССКЦ СО РАН с общей памятью HP DL980 G7 с четырьмя процессорами Intel E7-4870 (всего 40 ядер) и 1024 Гбайт RAM.

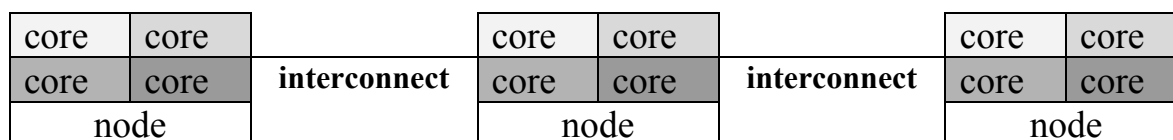


Рис. 5.25. Архитектура SMP-узлов кластера

OpenMP-реализации различных стратегий сборки невязки

Дополнительные временные затраты при применении технологии OpenMP для ВС с общей памятью связаны в основном с необходимостью синхронизации нитей (5.6). В нашем случае дополнительным по сравнению с последовательной программой будет время:

- на создание/уничтожение нитей;
- синхронизацию нитей при распределении работы тела цикла (определяется опцией `schedule` – стратегии `static`, `dynamic`, `guided`, `runtime`, `auto`);
- синхронизацию при выполнении глобальной редукции при расчете критерия останова итерационного метода решения СЛАУ (5.21) (например, методом Якоби);
- синхронизацию нитей внутри основного цикла при сборке невязки.

При сборке невязки по конечным элементам несколько нитей, возможно одновременно, могут обрабатывать один и тот же узел, но каждая в своем треугольнике (рис. 5.22); поэтому операция записи в элемент `xi[tr[i][1]]` или `xi[tr[i][j]]` в примере 5.17 в строках, помеченных в комментариях (1), является опасной. Эти операции необходимо защитить каким-либо механизмом синхронизации – директивами `omp atomic`, `omp critical` или замком `lock`.

Следовательно, при сборке невязки по конечным элементам необходимы дополнительные затраты на синхронизацию нитей. Расчеты, проведенные для модельных задач, показали, что синхронизация занимает до 40 % времени выполнения основного цикла, и эффективность распараллеливания получается крайне низкой – около 25 % (см. рис. 5.28).

При сборке невязки по узлам сетки каждый узел обрабатывается только одной нитью. В этом случае дополнительной синхронизации нитей не требуется, что дает явные преимущества этому подходу. Эффективность распараллеливания составляет около 90 % при использовании до 30 нитей и около 80 % при использовании более 30 нитей (одна нить на ядро).

Таким образом, *при сравнении эффективности распараллеливания различных стратегий сборки невязки в МКЭ с помощью технологии OpenMP явным преимуществом обладает сборка невязки по узлам сетки.*

МРІ-реализации различных стратегий сборки невязки

Используя явный параллелизм по данным, исходную расчетную область можно разбить на несколько частично перекрывающихся подобластей. Расчеты в каждой подобласти выполняются независимо друг от друга в рамках итерации решения СЛАУ (5.21). После каждой итерации необходимо проводить согласование данных в перекрытиях. В методе конечных элементов имеют место, по крайней мере, два варианта декомпозиции расчетной области.

1. *Декомпозиция с теневыми гранями.* Исходная область включает взаимно перекрывающиеся подобласти (рис. 5.26), ширина перекрытия определяется шаблоном дискретного аналога. При этом невязка в граничных точках подобласти i -го процесса насчитывается в подобластях соседних процессов. С учетом семиточечного шаблона и согласованности триангуляции в нашем случае достаточно перекрытия подобластей в два слоя расчетных точек. Поскольку алгоритм вычисления невязки требует хранения в каждой граничной точке подобласти семи коэффициентов матрицы жесткости, значения вектора решения текущей и предыдущей итераций и значения правой части, то избыточность хранения информации для p процессов составляет $40(p-1)N_{bnd} \text{ sizeof(double)}$ байт. Здесь через N_{bnd} обозначено количество точек на разрезе, sizeof(double) – количество байт, занимаемых переменной с плавающей точкой, хранимой с двойной точностью.

2. *Декомпозиция без теневых граней.* Исходная область разрезается на подобласти, пересекающиеся только по границам разреза (рис. 5.27). Для каждой граничной точки подобласти невязка насчитывается частично только по тем треугольникам, которые лежат в подобласти. При обмене данными после каждой итерации требуется дополнительное суммирование для значений невязки в граничных для подобласти точках.

Второй способ декомпозиции более экономичен по памяти и прост в программировании. Очевидно его достоинство для неструктурированных сеток, когда границы подобластей не являются последовательным множеством точек. Таким образом, декомпозиция без теневых граней выглядит несколько более привлекательной для наших целей.

Реализация параллельной программы для ВС с распределенной памятью осуществлялась на языке программирования Си с применением функций библиотеки передачи сообщений МРІ. Более подробно библиотека МРІ описана в гл. 6.

В рамках выбранной схемы распределения данных все процессы осуществляют одни и те же вычисления, но только над разными подобластями. Структура обменов также является однородной, за исключением первого и последнего процессов. После каждой итерации метода Якоби процесс выполняет обмен данными со всеми своими соседями, число которых определяется декомпозицией и не зависит от количества процессов, принимающих участие в расчете.

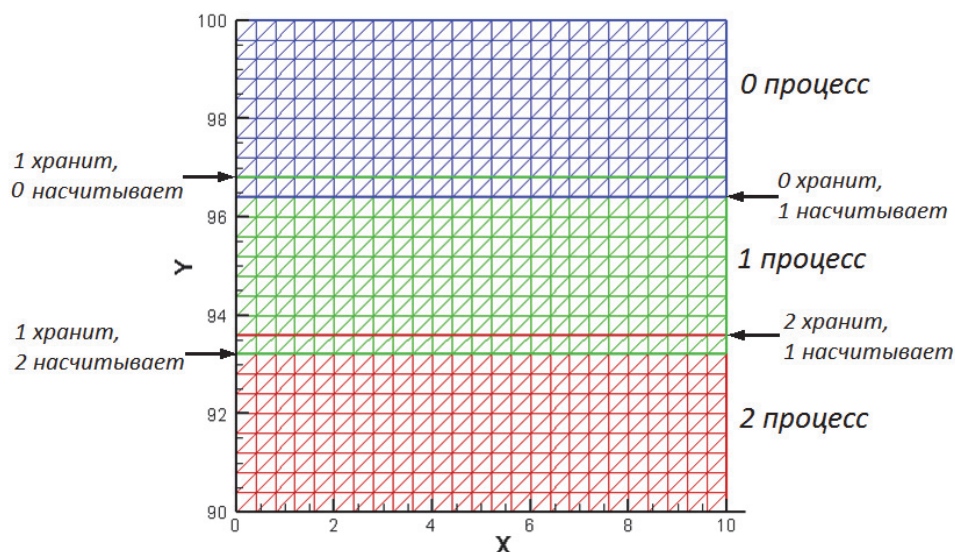


Рис. 5.26. Декомпозиция области с теневыми границами

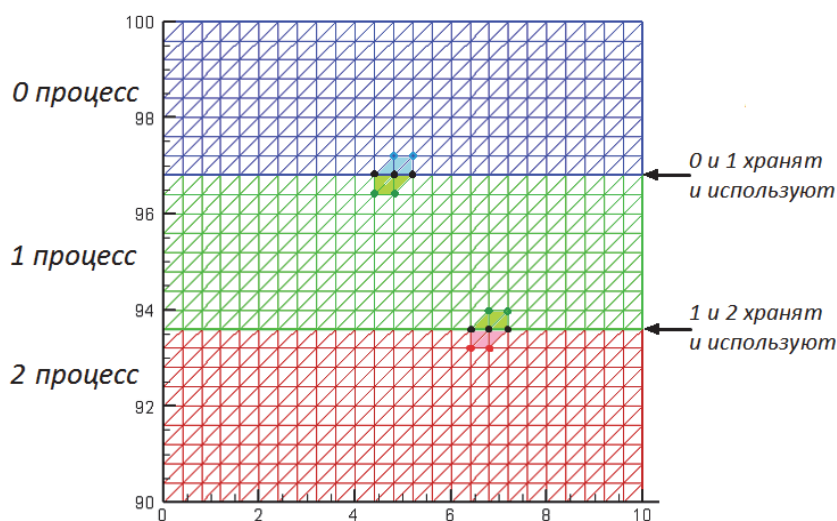


Рис. 5.27. Декомпозиция области без теневых границ

Приведем теоретические оценки потенциального ускорения, по возможности учитывая время, затрачиваемое при выполнении алгоритма на обмены.

Обозначим время выполнения одной арифметической операции через t_{op} , а время, затрачиваемое на пересылку одного значения, – через t_{comm} . Ясно, что последняя величина является, скорее, виртуальной характеристикой скорости пересылок, однако вполне подходит для наших теоретических оценок.

Пусть N_{nd} – общее количество узлов сетки расчетной области, s – количество операций на один расчетный узел на каждой итерации решения СЛАУ, k – количество итераций. Тогда время выполнения алгоритма на одном процессоре можно оценить следующим образом:

$$T_1 = skN_{nd}t_{op}.$$

Предположим, что при декомпозиции области нам удастся равномерно распределить весь объем вычислений по процессорам. В этом случае для времени выполнения алгоритма на p процессах (один процесс на один вычислитель) верна формула (5.4), откуда следует, что для получения хорошего ускорения MPI-программы необходимо, чтобы время, затрачиваемое на обмены, было значительно меньше времени, затрачиваемого на вычисления на p процессах.

Как следует из принятой нами схемы распределения данных, на каждой итерации решения СЛАУ (5.22) требуется выполнить следующие действия, порождаемые распределенностью данных: 1) глобальную операцию приведения для вычисления критерия останова итерационного процесса; 2) обмен значениями части вектора невязки в каждой точке разреза. Оценим затраты времени на эти операции.

1. Для вычисления критерия останова итерационного процесса на каждой итерации требуется найти глобальный максимум по всем процессам. Предположим, что реализация глобальных операций приведения в MPI выполняется по оптимальному алгоритму сдваивания, что дает следующую оценку времени, затрачиваемого на одну операцию приведения:

$$T_1^{comm}(p) = (t_{op} + t_{comm}) \log_2 p.$$

2. При использовании библиотеки MPI возможны два принципиально разных способа организации двухточечных обменов – с применением блокирующих или неблокирующих функций передачи данных. Наша реализация алгоритма допускает использование неблокирующих обменов, которые в этом случае, безусловно, выгоднее блокирующих, поскольку могут выполняться параллельно. Таким образом, время обменов не зависит от количества процессов, участвующих в расчетах, а зависит только от максимального количества соседей, которых имеет процесс. Наблюдается следующая оценка времени для обмена частями вектора невязки:

$T_2^{comm}(N_{bnd}) = \mu km N_{bnd} t_{comm}$, где m – количество значений которые необходимо передать соседнему процессу для одной точки разреза, μ – максимальное количество соседних процессов, с которыми происходит обмен.

В результате оценка потенциального ускорения нашего параллельного алгоритма для случая использования неблокирующих двухточечных обменов имеет вид

$$S_p(N) \approx \frac{1}{\frac{1}{p} + \frac{\log_2 p}{s N_{nd}} (1 + \kappa) + \frac{m\mu}{s} R \kappa}. \quad (5.25)$$

Из оценки (5.25) следует, что для достаточно мелких сеток потенциальное ускорение близко к линейному для достаточно большого диапазона количества процессов. Величина ускорения определяется двумя параметрами. Первый параметр $R = N_{bnd} / N_{nd}$ характеризует декомпозицию расчетной области. Приемлемое ускорение обеспечивается малостью R , поэтому при построении декомпозиций сложных вычислительных областей, наряду с требованием равенства вычислительной нагрузки на процесс, необходимо обеспечивать минимальную протяженность границы подобласти для каждого процесса. Второй параметр $\kappa = t_{comm} / t_{op}$ характеризует коммуникационную среду. Этот параметр описывает вычислительную сеть очень условно, но он показывает, что для приемлемого ускорения следует выбирать архитектуру кластера с небольшими значениями κ .

Отметим, что с учетом экономии памяти и легкости реализации на неструктурированных сетках преимуществом обладает декомпозиция без перекрытий.

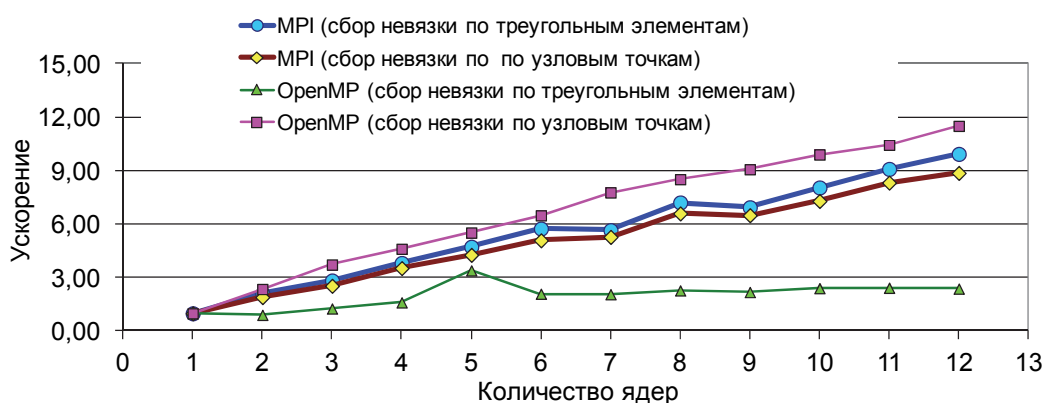


Рис. 5.28. График зависимости ускорения вычислений от количества используемых ядер для MPI- и OpenMP-версий программ

Как мы видим, при оценке потенциального ускорения не рассматривалось влияние способа сборки невязки, поскольку он не увеличивает время

работы параллельной программы, сохраняя все достоинства и недостатки работы с памятью последовательных версий. Поэтому для ВС с распределенной памятью остается актуальным вывод, сделанный для последовательной программы. *В распределенной среде преимущество получает реализация метода конечных элементов со сборкой невязки по элементам.*

Численные эксперименты показали преимущество сборки невязки по элементам с эффективностью параллельной реализации около 80 %. Рис. 5.29, в демонстрирует, что эта версия программы выполняется быстрее, чем MPI-версия при сборке невязки по узловым точкам, что объясняется более выгодным распределением памяти в первом случае. Особо следует отметить, что эта версия MPI-программы по времени выполнения выигрывает и по сравнению с самой быстрой OpenMP-версией.

Совмещение технологий MPI и OpenMP для SMP-узлового кластера

Кажется естественным для архитектуры SMP-узлового кластера совместить технологии OpenMP и MPI в одной параллельной реализации. Действительно, в рамках одного узла можно реализовать OpenMP-технологии, а между узлами – применить MPI.

Из (5.4) и (5.6) следует, что время выполнения гибридной OpenMP+MPI-программы, запущенной на p процессах по th OpenMP-нити, на каждый процесс (каждая нить выполняется на своем вычислителе) может быть оценено следующим образом:

$$T_{p \cdot th}^{SMP}(N) \approx \frac{T_1^{calc}(N)}{p \cdot th} + \frac{\gamma(th)T_1^{mem}(N)}{p \cdot th} + T_{th}^{synch}(th, N) + T_p^{comm}(p), \quad 1 < \gamma < th. \quad (5.26)$$

Из (5.25) видно, что гибридная технология будет выгодна только в том случае, если суммарное дополнительное время, затрачиваемое на синхронизацию OpenMP-нитей и MPI-коммуникацию гибридной программы, будет меньше времени, затрачиваемого на коммуникацию программы, использующей только технологию MPI:

$$T_{p \cdot th}^{comm} > \frac{\gamma(th) - 1}{p \cdot th} T_1^{mem}(N) + T_{th}^{synch}(th, N) + T_p^{comm}(p), \quad 1 < \gamma < th.$$

В численных экспериментах запускались два MPI-процесса на узел (по пять OpenMP-нитей на каждый MPI-процесс).

Из рис. 5.29 видно, что при сборке невязки по узловым точкам ускорение вычислений (MPI+OpenMP)-версии программы практически совпадает с линейным, а эффективность составляет около 100 %.

В случае сборки невязки по элементам эффективность параллельной реализации составляет всего 40 %, что ожидаемо ввиду затрат на синхронизацию нитей в OpenMP.

Результаты исследований показали, что *наиболее эффективной из рассмотренных параллельных реализаций алгоритма оказалась MPI+OpenMP-версия программы при сборке невязки по точкам сетки.* Следует отметить, что в то же время она является наиболее сложной в реализации (требует создания, хранения и обработки дополнительных структур).

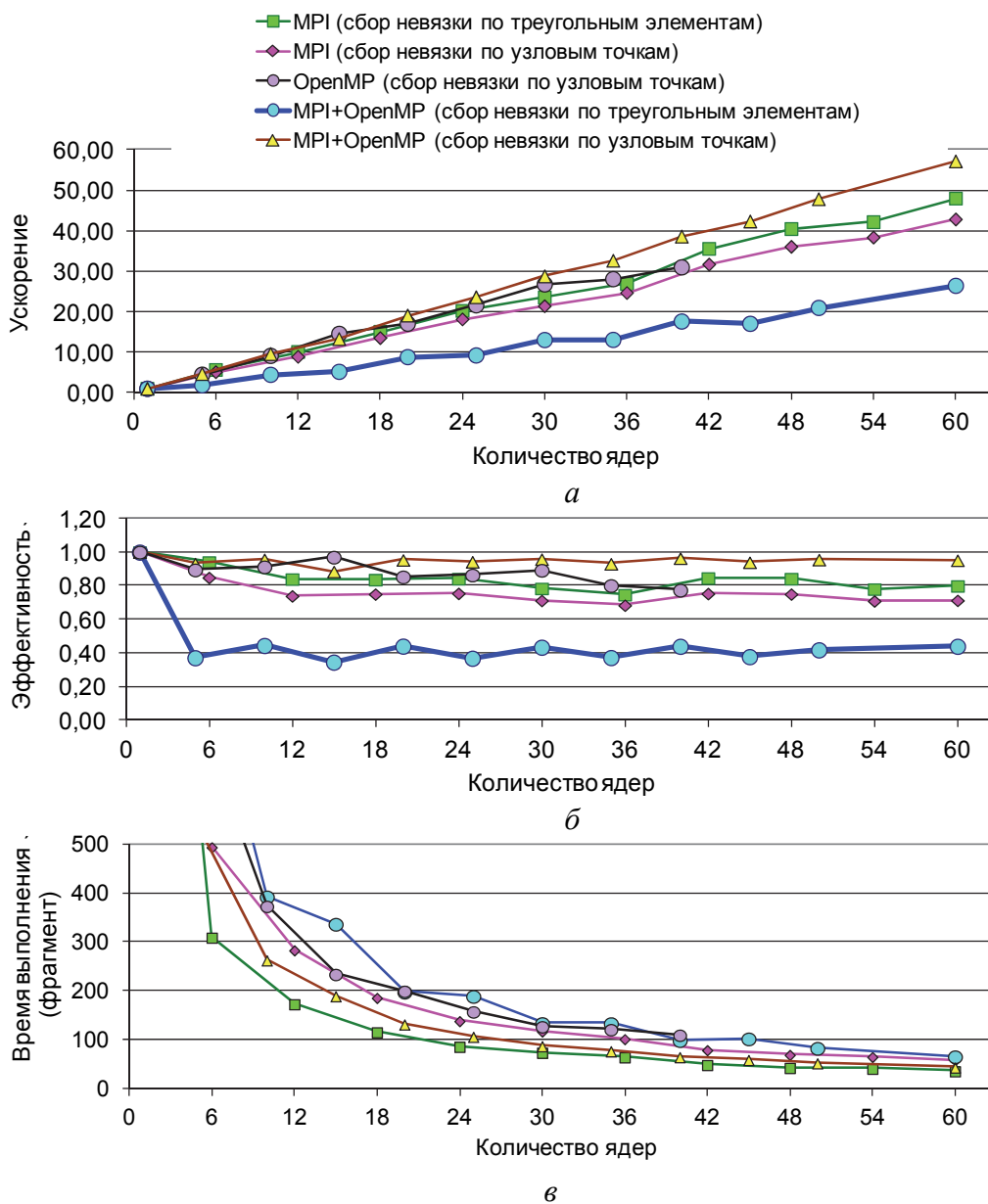


Рис. 5.29. Графики зависимости ускорения вычислений (а), эффективности (б) и времени выполнения (в) от количества используемых ядер для MPI-, OpenMP- и MPI+OpenMP-версий программ

Надо также понимать, что *наименьшее время выполнения показывает MPI-версия со сборкой невязки по треугольным элементам* (рис. 5.29).

Контрольные вопросы и задания

1. В чем заключается специфика программирования для параллельных вычислительных систем с распределенной памятью?
2. Чем механизм передачи сообщений отличается от механизма передачи данных через общую память?
3. Объясните особенности, черты сходства и различия синхронной и асинхронной передачи сообщений.
4. Какие методы взаимодействия параллельных процессов используются при передаче сообщений?
5. В чем заключаются механизмы удаленного вызова процедур и рандеву? Каковы основные сходства и различия этих механизмов и передачи сообщений?
6. Какие методы планирования процессов целесообразно применять для вычислительных систем с распределенной памятью?
7. Опишите подход к изучению информационных зависимостей алгоритма на основе графа «операции-операнды». Определите понятия «вершина ввода», «входные и выходные вершины», «ярус», «ширина яруса». Ответ проиллюстрируйте примером.
8. Сформулируйте основные теоретические оценки и связи между временем выполнения алгоритма на одном процессоре, минимально возможным временем выполнения параллельного алгоритма при использовании неограниченного количества процессоров и временем выполнения параллельного алгоритма на конечном числе процессоров. Ответ проиллюстрируйте примером.
9. Сформулируйте подходящий для распараллеливания алгоритм вычисления площади прямоугольника по заданным координатам его диагонали. Постройте граф его информационных зависимостей, вычислите T_1 , T_∞ , T_p .
10. Определите понятие ускорения, получаемого при использовании параллельного алгоритма для p процессоров, по сравнению с последовательным вариантом выполнения вычислений. В каком случае говорят о линейном ускорении?
11. Каковы могут быть причины возникновения сверхлинейного ускорения параллельного алгоритма?
12. Определите понятие эффективности использования параллельного алгоритма процессоров. В каком случае говорят о хорошей масштабируемости параллельного алгоритма?
13. Сформулируйте закон Амдала, его следствие и закон Гюставсона — Барсиса. В чем сходства и различия оценок ускорения в этих законах?

14. Проиллюстрируйте понятия ускорения и эффективности выполнения параллельного алгоритма на примере задачи о суммировании n чисел.

15. Расскажите о возможных решениях проблемы распределенности данных в ВС с распределенной памятью. Ответ проиллюстрируйте примерами.

16. Приведите общую схему алгоритма пульсации. Приведите задачу, для параллельного решения которой хорошо подходит этот алгоритм.

17. Объясните суть коллективной операции «синхронизация с барьером». Укажите несколько путей ее возможной реализации на основе механизма передачи сообщений.

18. Какие глобальные функции связи вы знаете? Ответ проиллюстрируйте примерами, в которых желательно их использование.

19. Какие глобальные операции приведения вы знаете? Ответ проиллюстрируйте примерами, в которых желательно их использование.

20. Что такое латентность и пропускная способность сети?

21. Опишите основные подходы к оценке времени передачи сообщения на кластерных ВС.

22. Расскажите о возможных решениях проблемы балансировки нагрузки и ее значении для эффективности параллельного алгоритма.

23. В чем суть параллельного алгоритма «скалярных произведений» вычисления произведения матрицы на вектор?

24. В чем суть параллельного алгоритма «линейной комбинации» вычисления произведения матрицы на вектор?

25. Опишите параллельный алгоритм вычисления произведения матриц в топологии связей «кольцо».

26. Опишите параллельный алгоритм вычисления произведения матриц в топологии связей «2D-решетка».

27. Опишите параллельный алгоритм вычисления произведения матриц в топологии связей «3D-решетка».

28. Каковы способы распараллеливания алгоритма решения СЛАУ методом Гаусса.

29. Напишите схему параллельного алгоритма решения СЛАУ методом простой итерации, используя нотацию для обменов «точка – точка» и коллективных операций, введеную в главе.

30. Опишите основные моменты параллельного алгоритма решения задачи Дирихле для уравнения Пуассона методом Якоби.

Глава 6 | ОСНОВЫ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ MPI

В главе рассмотрены основы программирования для ВС с распределенной памятью с помощью библиотеки MPI. Описана архитектурная парадигма MPI, ее связь с крупноблочным распараллеливанием. Обсуждаются вопросы организации вычислений (использование коммутаторов, производных типов, виртуальных топологий) и взаимодействий процессов (двухточечные и коллективные обмены, операции приведения и барьерной синхронизации). Отметим, что, хотя в тексте и приведен синтаксис большинства функций MPI для языка Си, главу не следует рассматривать как описание стандарта MPI. Все вводимые концепции, понятия и методы проиллюстрированы примерами.

6.1. Архитектурная парадигма MPI

Рассмотрим «задачу о землекопах» на современный лад. Пусть трех рабочих нужно проинструктировать, как выкопать и обустроить некоторым образом яму. Причем их работу надлежит организовать так, чтобы ни одному не пришлось действовать «по обстановке»: и личные действия каждого, и взаимодействия в коллективе должны быть строго расписаны по шагам. Также пусть каждый из рабочих ничем не отличается от другого в части способностей к работе любого рода, будем называть их «первый», «второй» и «третий». Каким образом можно поставить им такую задачу?

Оценив общий объем работ, разделим его на некоторые крупные этапы (декомпозиция на основе логики действий), например: копание ямы, обустройство ямы, уборка мусора. Каждый этап разделим на три равные и по возможности непересекающиеся части так, чтобы можно было работать втроем одновременно (декомпозиция на основе логики данных, исключение или минимизация межзадачных связей). После этого можно составить три различных руководства (многопоточное приложение), подписав на них «Первому», «Второму», «Третьему», и присесть в тени, чтобы любоваться слаженным трудом нашего коллектива, задумавшись об эффективности собственного труда. А если необходимо составить руководство по обустройству ямы для неопределенного количества таких рабочих, скажем, от двух до двадцати? Ясно, что, помимо времени и сил рабочих, нам нужно также беречь и свои силы и время, следовательно, вместо

двадцати разных вариантов указаний можно составить один, универсальный, вариант, основанный на следующих принципах:

1. Наличный объем работ разбивается на такое (неизвестное заранее) количество частей, сколько рабочих выделено для его исполнения.

2. В одном руководстве приведены как указания, относящиеся ко всем исполнителям одновременно, так и касающиеся только «первого», «второго» и т. п. При выборе своей части работы каждый исполнитель руководствуется собственным номером и общим количеством исполнителей, эта информация всем становится известна перед стартом. Проще говоря, рабочие в самом начале «рассчитываются по порядку».

3. В процессе работы рабочие обмениваются между собой информацией и материалами с помощью некоторых приемов и средств установленного образца (механизм «передачи сообщениями»).

Описанные принципы представляют собой архитектурную основу стандарта MPI. Для запуска программы на любом количестве процессоров пишется *одна* программа, в которой реализуются ветвления вида «если я – номер второй, то выполнить этот блок, иначе – пропустить», «если я – номер первый, то подметать, иначе – вместе со всеми собирать мусор» и т. д. Избыточность этого подхода окупается его масштабируемостью и гибкостью, а неэффективность в большинстве случаев оказывается кажущейся.

В настоящее время MPI (Message Passing Interface) стал наиболее распространенной технологией программирования параллельных компьютеров с распределенной памятью. В его основу положен базовый способ взаимодействия параллельных процессов в таких системах – передача сообщений. Современные реализации чаще всего соответствуют стандарту MPI версии 1.1, разработанному в 1993–1995 годы группой MPI Forum, в состав которой входили представители академических и промышленных кругов [88]. Хотя в 1998 году и появился стандарт MPI-2.0, значительно расширивший возможности первой версии, до сих пор этот вариант MPI не получил широкого распространения. Поэтому в этой главе обсуждается в основном реализация MPI, соответствующая версии 1.1¹. Детальное обсуждение технологии MPI содержится в монографиях [1, 4, 11, 14, 25–27, 32, 63].

Дотошность и последовательность, проявленная разработчиками стандарта, способна отпугнуть начинающего от его изучения: настолько велико количество типов сообщений и вариантов их комбинирования, а также других возможностей. Конечно, никто не принуждает программиста пользоваться ненужными ему возможностями, и ничто не мешает составить такую параллельную программу с использованием MPI, которая

¹ Полные тексты стандартов размещены в Интернете по адресу <http://www-unix.mcs.anl.gov/mpi>.

будет работать только на конкретном количестве узлов или же на некотором их множестве (например, на четном количестве узлов или на количестве узлов, равном квадрату любого натурального числа).

Особо следует подчеркнуть принципиальную неразличимость и взаимозаменяемость исполнителей. Никогда не известно перед началом работы программы, на каких именно компьютерах кластера она будет исполняться, каким номером будет считать себя каждый из них. Стандарт гарантирует только наличие заказанного количества исполнителей. Такое ограничение напрямую связано с тем, что любой суперкомпьютер является устройством коллективного пользования, т. е. на нем работают и запускают свои задачи множество различных людей. Естественно, что постоянный конфликт, в котором находятся их интересы, подлежит регулированию. Это делается с помощью некоторой системы запуска параллельных задач. Стандарт MPI почти не касается этого аспекта, предоставляя алгоритм и способ работы такой системы на откуп разработчикам конкретных реализаций и специфицируя только синтаксис некоторых команд общего характера. Однако именно принцип неразличимости исполнителей, заложенный в стандарте, делает возможной более или менее эффективную работу этих систем.

6.2. Обрамляющие и информационные функции MPI

Для организации работы простейшей программы на языке Си в среде MPI достаточно очень небольшого набора функций.

Любая программа на языке Си, использующая библиотеку MPI, должна подключить заголовочный файл `<mpi.h>`. Функция `main()` должна в обязательном порядке получать данные из командной строки. Дело в том, что загрузчик MPI сообщает программе некоторую информацию, необходимую для поддержки среды. В примере 6.1 содержится код простейшей программы, использующей библиотеку MPI. В примере каждый процесс после запуска узнает общее количество запущенных процессов, свой номер в этой группе и отправляет на консоль сообщение об этом. Дополнительно процесс с номером 0 выводит на консоль параметры, которые ему передал загрузчик MPI через командную строку.

Опишем основные функции, которые принято называть обрамляющими, поскольку они как бы заключают в рамку программу, использующую MPI.

```
int MPI_Init (int* argc, char** argv);
```

Функция инициализации параллельной части программы. В конец командной строки программы MPI-загрузчик добавляет ряд информационных параметров, которые требуются `MPI_Init()`.

До вызова `MPI_Init()` нельзя вызывать ни одной функции MPI. При повторном вызове функции никакие действия не выполняются, происходит немедленный возврат из функции. Все другие функции MPI могут вызываться только после вызова `MPI_Init()`.

```
int MPI_Finalize(void);
```

Функция обеспечивает завершение параллельной части приложения. Все последующие обращения к любым функциям MPI, в том числе к `MPI_Init()`, запрещены. К моменту вызова процессом `MPI_Finalize()` все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Следующие три функции – информационные, сообщающие процессу некоторые внешние условия, в которых он вынужден работать.

```
int MPI_Comm_size(MPI_Comm comm, int* size);
```

Функция определяет в параметр `size` общее число параллельных процессов в области связи, ассоциированной с коммуникатором `comm`. Подробнее понятия области связи и коммуникатора обсуждаются ниже. Сейчас заметим только, что при запуске MPI-программы создается область связи, объединяющая все запущенные процессы в группу с предопределенным коммуникатором `MPI_COMM_WORLD`.

```
int MPI_Comm_rank(MPI_comm comm, int* rank);
```

Функция возвращает в `rank` номер процесса, вызвавшего ее, в области связи, ассоциированной с коммуникатором `comm`. Значение, возвращаемое по адресу `&rank`, лежит в диапазоне от 0 до `size-1`, где `size` – размер группы, связанной с коммуникатором `comm`.

```
double MPI_Wtime(void);
```

Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Гарантируется, что этот момент не будет изменен за время существования процесса.

6.3. MPI и крупноблочное распараллеливание

Стандарт MPI возник не на пустом месте – фундаментом технологии является архитектура массивно-параллельных ЭВМ (МРР), для работы на которых она в целом и предназначена. Напомним, что МРР-система представляет собой множество вычислительных узлов, каждый из которых

снабжен своей локальной памятью, однако узлы не имеют прямого доступа к локальной памяти друг друга. Связь между ними осуществляется с помощью некоторой коммуникационной сети. Обмен между узлами может быть организован множеством способов, наиболее общим является передача сообщений, которая бывает либо типа «точка – точка» (т. е. один узел – другому), либо «коллективного» типа (с участием более чем двух узлов). Именно такой модели соответствует подход MPI к организации взаимодействий между узлами, по сути являясь тем инструментом, с помощью которого крупноблочное распараллеливание реализуется на практике.

Пример 6.1. Использование обрамляющих и информационных функций MPI

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char **argv){
    int size, rank, i;
    MPI_Init( &argc, &argv ); //инициализация MPI-библиотеки
    //Определение количества запущенных процессов
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    //Определение процессом своего номера (ранга)
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    //Только процесс с рангом 0 выводит информацию на печать
    if( rank==0 ) printf("Total processes count = %d\n", size);
    //Все ветви сообщают свой ранг
    printf("Hello! My rank in MPI_COMM_WORLD = %d\n", rank);
    //Точка синхронизации (см.тему «Коллективные взаимодействия»)
    MPI_Barrier( MPI_COMM_WORLD );
    //Процесс с рангом 0 выводит на печать
    //аргументы командной строки, которая может содержать
    //параметры, добавляемые загрузчиком mpirun
    if (rank == 0)
        for(i=0; i<argc; i++)
            printf("0: Command line - %d: \"%s\"\n",i,argv[i]);
    MPI_Finalize(); //закрытие MPI-библиотеки
    return 0;
}
```

Ранее мы «интуитивно» раздали нашим рабочим крупные и по возможности непересекающиеся части общего дела: было бы очевидной практической нелепостью предлагать им копать яму, держась втроем за одну лопату. Но для этого нам пришлось разделить области приложения их усилий. Аналог такого подхода в вычислениях и называется блочным распараллеливанием: каждый вычислительный узел получает в свое распоряжение некоторую часть данных задачи и оперирует ими, при необходимости обмениваясь с остальными недостающими данными. Очевидно, что чем

больше эти объемы и чем реже происходят обмены, тем эффективнее процесс распараллеливания. В гл. 5 уже обсуждались основные способы балансировки нагрузки.

Можно сказать, что MPI пригоден для распараллеливания определенного (весьма обширного) класса вычислительных задач, который характеризуется 1) необходимостью обработки больших объемов данных и 2) возможностью обрабатывать различные данные одновременно (хотя бы частично), т. е. является подходящим средством для согласованного параллельного программирования в средах с распределенной памятью. Отметим, что согласованность следует в таких задачах понимать как разновидность параллельности по данным – несколько независимых процессов выполняют, скорее всего асинхронно, один и тот же код, каждый над своим набором данных.

Подход к написанию параллельных приложений с помощью библиотеки MPI повторяет общий подход ДСС (декомпозиция – связь – синхронизация), представленный в гл. 1 и уже изученный нами при проектировании параллельных приложений для архитектур с общей памятью. Применительно к задачам, параллельным по данным, алгоритм можно сформулировать следующим простым образом: 1) разрезать область данных на блоки; 2) распределить части данных на вычислительные узлы; 3) организовать обработку данных (счет), по необходимости используя обмены; 4) собрать результаты расчетов.

Рассмотрим некоторые подробности каждого из этих этапов, по ходу изложения вводя соответствующие инструменты MPI.

Декомпозиция по данным

Пусть имеется N вычислительных узлов и некоторое количество данных. Предположим для определенности, что данные представляют собой значения в узлах сетки, а область будет двумерной и однородной.

Такое множество данных можно разделить на блоки многими способами. Рассмотрим только разбиения на связные (не содержащие разрывов внутри) или даже просто выпуклые блоки равного размера (в данном случае – содержащие равное количество точек сетки).

Наиболее легкий путь – это просто разрезать область на полосы (ленты) и распределить их по узлам. Его преимущество – простота, приводящая к следующим обстоятельствам:

- у каждого узла всего два соседа, чьи номера легко вычисляются;
- в зависимости от языка (Си или Фортран) можно разрезать область по строкам либо столбцам в соответствии со способом хранения в памяти массивов, что впоследствии облегчит обмены;

- добиться приблизительного равенства количества данных на узлах несложно, даже если соответствующая характеристика области (длина или ширина) не делится нацело на количество вычислительных узлов.

Эти неоспоримые преимущества делают данный подход очень популярным. Его основной недостаток общего характера состоит в том, что *периметр* каждой области далек от минимально возможного.

Поскольку в большинстве практических сеточных (и не только) задач необходимость в обменах с соседями возникает именно на границах областей, то от периметра зависит общий объем данных, которые придется принять и передать данному узлу.

Простейший способ привести периметр к *практическому* минимуму — это использовать не ленточное, а квадратное разбиение: при равной площади периметр квадрата меньше периметра прямоугольника. Кроме того, квадратное разбиение выгоднее с точки зрения борьбы с известным эффектом деградации эффективности (уменьшением параллельной эффективности с ростом количества узлов при сохранении размера задачи): для ленты время совершения обменов не зависит от количества узлов, а для квадрата — уменьшается пропорционально $1/\sqrt{p}$. Однако и сложность конструирования такого разбиения растет.

Почему же не пойти дальше, к минимуму отношения площади и периметра — кругу? Конечно, кругами нельзя покрыть плоскость, но это можно сделать пятиугольниками. Однако мы недаром подчеркнули выше практичность минимума квадратного разбиения. Пятиугольник непрактичен по многим причинам. Прежде всего, наша сетка дискретна и прямоугольна, следовательно, пока границы областей совпадают с линиями сетки, мы встречаем ожидаемое согласие между аналитическими оценками тех же периметров и площадей и поточечной реальностью. Как только границы областей находятся под углом к пространственной структуре сетки, начинаются расхождения, которые могут не только свести на нет предполагавшиеся выгоды, но и сильно увеличить время отладки программы.

Усложним теперь наши условия: пусть сетка станет неоднородной, имеющей сгущения и разрежения. В таких условиях все будет зависеть от рисунка изолиний густоты сетки. Например, в некоторых задачах, моделирующих слоистые среды, бывает естественным изменять густоту сетки согласно расположению соответствующих слоев. В других задачах сетка может сгущаться к границам области или же к середине. Во всех случаях при разбиении на блоки необходимо добиваться соблюдения следующих условий:

- равенства площадей (объемов — в трехмерном случае) в смысле количества точек сетки для обеспечения равномерной вычислительной нагрузки на узлы;

- уменьшения количества соседей для каждого узла;
- увеличения отношения площади области к ее периметру.

Может оказаться, что некоторые из этих пунктов противоречат друг другу. В результате поиск баланса, оптимального с точки зрения главного критерия – времени счета, составляет плохо формализуемую, творческую часть процесса распараллеливания.

Распределение данных по узлам

Итак, мы определили, как именно должна быть разрезана область на p подобластей. Теперь необходимо проинициализировать области на узлах начальными значениями (или, что хуже, результатами предыдущей счетной итерации), а по окончании счета – выгрузить с узлов посчитанные значения.

Кластерные системы обычно предоставляют доступ к домашнему каталогу пользователя со всех вычислительных узлов; местонахождение этого каталога зависит от «бюджетности» кластера и может варьироваться от обычного жесткого диска или файлового массива до специализированного файлового хранилища. Если размеры данных действительно невелики, то каждому вычислительному узлу будет совсем не трудно открыть нужный файл и прочесть или записать его. Однако по мере роста количества данных, подлежащих чтению или записи, ситуация ухудшается стремительно и может стать катастрофической, особенно если попутно растет количество процессов, желающих сделать это одновременно, так как производительности дисковых интерфейсов и коммуникационных сред, которыми эти интерфейсы соединены с узлами, скорее всего, являются величинами одного порядка. Прибавив к этому тот факт, что на кластере работает не только наша программа и не только она может занимать файловый сервер и канал к нему, легко попасть в ситуацию, когда сохранение результатов счета на диск окажется занятием намного более долгим, чем сам счет.

Есть вещи, которые не стоит делать в любом случае. Первое, чего следует избегать, – чтения или записи одного файла многими процессами одновременно. Файловая и сетевая подсистемы, скорее всего, справятся с этой задачей, но производительность существенно пострадает. Не намного лучше одновременные чтение/запись в большое количество файлов. Помимо потери производительности в узких местах при передаче данных, в такой ситуации возникнет необходимость в постобработке созданных файлов, например, в нетривиальной их склейке. Это может оказаться неожиданно затруднительной задачей, если, к примеру, результирующий файл не помещается в памяти на одном узле или если он вместе с исходными файлами, удвоив собой их объем, превысит дисковую квоту. Все это говорит о том, что работа с загрузкой и выгрузкой данных должна быть

тщательно и аккуратно продумана еще на этапе проектирования архитектуры.

Какими средствами можно воспользоваться? Во-первых, стандарт MPI-2 (расширение стандарта MPI) предоставляет в распоряжение пользователя средства работы с файлами. Использовать их или нет, следует решать, сообразуясь с архитектурой кластера и самой задачи.

Во-вторых, иногда на выходе оказываются нужны только качественные результаты, например, вместо точных значений неизвестных – построенные по этим значениям картинки (занимающие намного меньше места). Для генерации растровых картинок можно, например, использовать популярную библиотеку Boutell GD2.

В-третьих, как уже обсуждалось ранее, возможным выходом является парадигма «управляющий – рабочий», т. е. выделение отдельного узла-диспетчера, занятого исключительно обслуживанием ввода-вывода и прочими вещами, не имеющими прямого отношения к *параллельному* счету. Например, управляющему процессу можно поручить следующие действия:

1. Загрузка данных из хранилища (линейным чтением большими блоками с максимальной скоростью) и раздача их по вычислительным узлам (используя высокоскоростную сеть передачи данных).

2. Элементы синхронизации и контроля процесса счета, проверка глобальных критериев остановки, диспетчеризация, мониторинг, журналирование.

3. Сбор (по сети передачи данных) результатов счета, организация временного хранения собранных данных, преобразование промежуточных данных (непараллельный постпроцессинг, визуализация, сжатие и т. п.).

4. Выгрузка промежуточных и финальных результатов на диск.

Преимущества использования отдельного узла для решения задач 3 и 4 состоят в том, что эти задачи могут решаться независимо от вычислительного процесса, в течение счетной итерации. Лишь бы узел-диспетчер успевал за счетом. Если не успевает один, можно использовать два или более. Основной же недостаток уже обсуждался в гл. 4: при всей полезности узла-диспетчера он не занят вычислительной работой и при честном подсчете эффективности портит цифры.

Подсчет эффективности работы программы, решающей реальную задачу, – проблема, требующая еще некоторого пояснения. Напомним, что эффективность – это отношение ускорения к количеству узлов, на котором оно достигнуто. Однако возникает вопрос, в сравнении с чем вычислять ускорение? «Честный» подсчет предполагает сравнение с *наилучшей* последовательной программой, решающей ту же задачу, что и наша, на одном узле данного кластера. Вместо этого в большинстве случаев сопоставляются наша параллельная программа с нашей же программой,

запущенной на одном узле – если она, конечно, допускает такой режим запуска. Наряду с этой уловкой часто не учитывают время на загрузку-выгрузку данных или наличие несчитающих узлов. Отметим, что оценки эффективности обычно малоинтересны как таковые. Интересна для оценки перспективности программы либо подхода, реализованного в ней, динамика эффективности с ростом количества задействованных узлов или с ростом размера данных задачи, а эта асимптотика проявляет себя одинаково вне зависимости от учета тех или иных констант.

6.4. Организация вычислений

Одна из главных целей библиотеки MPI – организация обменов данными между процессами в ходе расчетов. Рассмотрим средства, предлагаемые библиотекой MPI для эффективной реализации всех этапов обмена.

Основные вопросы, на которые нужно найти ответ, таковы: на передающей стороне – куда, что и как передать; на принимающей стороне – от кого, что и как принять.

Средства адресации пересылок

На понятийном уровне отвечать на вопрос «кому послать сообщение?» легко: соседу слева, соседу справа, всем счетным узлам... Однако на практике для этого нужно знать номера этих соседей в некоторой группе MPI-процессов. Номера, которые присваиваются процессам в группе, создаваемой по умолчанию, могут мало соответствовать реальной топологии, например, квадратному разбиению с узлом-диспетчером.

Основные средства, предлагаемые MPI для упрощения задачи адресации пересылок, – это создание *отдельных групп процессов* и *новых топологий*.

Создание отдельных групп процессов. Полезность этой возможности на практике зачастую недооценивается. Между тем, освоив её, мы сможем применять богатый арсенал групповых операций к ограниченному числу процессов, сильно упрощая и оптимизируя код. Упрощая – потому что вместо циклов и последовательностей приемов/передач можно будет написать один вызов. Оптимизируя – потому что групповые обмены, наверняка, реализованы внутри MPI-системы лучше, чем это удалось бы нам самостоятельно.

Некоторые функции и понятия MPI обсуждаются далее в этой главе, описание более специальных функций выходит за рамки нашего подробного обсуждения и их следует искать в стандарте MPI [79, 88] и, например, в монографиях [13, 14, 26, 32].

Итак, процессы внутри приложения объединяются в *группы*. Один и тот же процесс может входить в несколько различных групп. Каждый процесс может узнать у библиотеки связи свой номер (*ранг*) внутри группы с помощью функции `MPI_Group_rank()` или, если с группой ассоциирован коммуникатор, с помощью функции `MPI_Comm_rank()`. В MPI принято в зависимости от номера поручать процессу выполнять соответствующую часть программы. В распоряжение программиста предоставлен тип `MPI_Group` и набор функций, работающих с переменными и константами этого типа.

Согласно концепции MPI, после создания группу нельзя дополнить или усечь – под требуемый набор процессов можно создать только новую группу на базе существующей.

При запуске MPI-приложения все процессы помещаются в общую стартовую *область связи* приложения. При необходимости процессы могут создавать новые области связи на базе существующих. Все области связи имеют независимую друг от друга нумерацию процессов. Абонентами одной области связи являются *все* задачи либо *одной*, либо *двух* групп.

Область связи – нечто абстрактное: в распоряжении программиста нет типа данных, описывающего непосредственно области связи, как нет и функций по управлению ими. Программе пользователя в распоряжение предоставляется *коммуникатор* – описатель области связи, для которого предусмотрен специальный тип `MPI_Comm`, описывающий некоторую распределенную структуру данных. Области связи автоматически создаются и уничтожаются вместе с коммуникаторами. Для стартовой области связи автоматически создается коммуникатор с идентификатором `MPI_COMM_WORLD`.

Для одной и той же области связи может существовать несколько коммуникаторов, в результате приложение будет работать с ней как с несколькими разными областями.

Многие функции MPI имеют среди входных аргументов коммуникатор, ограничивающий сферу их действия той областью связи, к которой он прикреплен. Коммуникаторы являются «несообщающимися сосудами», т. е. если данные отправлены через один коммуникатор, то процесс-получатель сможет принять их только через этот же самый коммуникатор, но ни через какой-либо другой.

Самым общим способом создания коммуникатора является связывание его с группой процессов. При этом следует придерживаться следующей технологии (подробное описание функций следует смотреть в стандарте):

1. Функцией `MPI_Comm_group(comm, &group)` определяется группа `group`, связанная с существующим коммуникатором `comm`.

2. На базе существующих групп функциями семейства `MPI_Group_*()` создаются новые группы с нужным набором ветвей (в MPI

определено семь функций конструирования групп, а также несколько информационных функций).

3. Для итоговой группы процессов `group_end` коммуникатора `comm` функцией `MPI_Comm_create(comm, group_end, comm_new)` создается коммуникатор `comm_new`.

4. Все описатели созданных в п. 2 групп очищаются вызовами функции `MPI_Group_free(&group)`.

Заметим, что функция `MPI_Comm_create()` является коллективной, т. е. должна быть вызвана *всеми процессами*, входящими в область связи коммуникатора `comm`. В примере 6.8 продемонстрировано применение описанной технологии для организации независимых коллективных обменов в двух непересекающихся группах процессов.

Существует также два способа создания коммуникатора непосредственно из уже существующих.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *comm_new);
```

Функция копирует существующий коммуникатор `comm` в новый коммуникатор `comm_new`.

```
int MPI_Comm_split(MPI_Comm comm, int split,
                   int rank, MPI_Comm *comm_new);
```

Функция расщепляет группу, связанную с коммуникатором `comm`, на непересекающиеся подгруппы. Количество подгрупп определяется количеством различных значений параметра `split`. Каждая подгруппа содержит все процессы, у которых значение `split` в вызове оказалось одинаковым. Внутри каждой подгруппы процессы ранжированы в порядке, определенном значением аргумента `rank`, со связями, отличными от их рангов в старой группе. Новая область связи создается для каждой подгруппы. Каждый процесс получает в параметре `comm_new` новый коммуникатор, связанный с одной из подгрупп. Процесс может определять значение `split` равным константе `MPI_UNDEFINED` для процессов, не принадлежащих любой новой группе, тогда в `comm_new` возвращается константа `MPI_COMM_NULL`.

Функции `MPI_Comm_dup()` и `MPI_Comm_split()` являются коллективными, следовательно, должны быть вызваны всеми процессами, входящими в область связи коммуникатора `comm`. Перед вызовом функции `MPI_Comm_split()` в каждом процессе должны быть определены значения переменных `split` и `rank`, соответствующие целям создания новых областей связи. Значение `split` должно быть неотрицательным. Возвращаемый в `comm_new` описатель будет принимать в разных процессах разные значения (всего столько разных значений, сколько создано подгрупп).

Посредством создания групп процессов и областей связи над этими группами процессы внутри параллельного MPI-приложения могут объеди-

няться в коллективы для решения промежуточных задач. Пользуясь описанием конкретной области связи (коммуникатором), процессы гарантированно ничего не примут извне подгруппы и ничего не отправят наружу, при этом они могут продолжать пользоваться любым другим имеющимся в их распоряжении коммуникатором для пересылок вне подгруппы.

Функции коллективных обменов создают дубликат полученного в соответствующем аргументе коммуникатора и передают данные через него, не опасаясь, что их сообщения будут случайно перепутаны с сообщениями функций обменов типа «точка – точка», распространяемыми через оригинальный коммуникатор. Программист с этой же целью в разных частях кода может передавать данные между процессами через разные коммуникаторы, один из которых создан копированием другого.

Имеется два типа коммуникаторов.

Интракоммуникатор используется для связи внутри отдельной группы процессов. Он описывает группу процессов и топологию, отражающую логическое расположение процессов в группе. Все основные функции MPI либо не различают типа коммуникатора, либо требуют интракоммуникатора, более того, этот тип коммуникаторов достаточен для реализации большинства алгоритмов.

Интеркоммуникатор используется для точечной связи между двумя непересекающимися группами процессов. Никакая топология не связывается с коммуникаторами этого типа.

В заключение можно отметить, что область связи, описываемая специальной распределенной структурой данных «коммуникатор», является обобщением и реализацией понятия канала, введенного в гл. 5.

Проиллюстрируем все сказанное на примере 6.2, в котором на основе существующего коммуникатора `MPI_COMM_WORLD` создается два новых:

- 1) для узла с номером 0, который выделяется под задачи диспетчеризации;
- 2) для всех остальных узлов, которые предназначены для вычислений.

Номер процесса в группе, ассоциированной с коммуникатором `MPI_COMM_WORLD`, хранится в переменной `id`.

Строка, помеченная (1), указывает системе, что происходит разбиение группы `MPI_COMM_WORLD` на несколько новых; при этом второй параметр определяет, в какую новую группу должен попасть данный процесс. Будет создано столько групп, сколько разных значений этого параметра окажется во всех процессах. В нашем случае это, как нетрудно видеть, значения 0 для нулевого процесса и 1 для всех остальных. Второй же параметр указывает желаемый номер процесса в новой группе. Мы не захотели заказать их явно (все процессы передали одинаковый номер – ноль), так что система сама раздаст нашим процессам новые номера, основываясь на

старой нумерации: в нашем случае номером каждого процесса в `Comm_Group` должно стать значение `id-1`.

Итак, процессы поделены на две группы, а в каждом процессе стал доступен новый коммуникатор `Comm_Group`. Упоминание его вместо `MPI_COMM_WORLD` в любом последующем MPI-вызове приведет к тому, что вызов будет касаться только тех процессов, которые входят в ту же группу, что и он сам. Так, процесс-диспетчер, используя `C_Group`, не сможет увидеть никого, кроме себя, а счетные узлы в рамках `C_Group` смогут обмениваться между собой, не включая в обмен диспетчера. Строки (2) и (3) сообщают процессам их номера в новой группе и размер этой группы.

В примере 6.2 предполагается, что все p процессов, предназначенных для вычислений, разбиты на $l1$ групп по $l2$ процесса в каждой ($l1 \times l2 = p$). Строки (4)–(7) определяют номера соседей каждого процесса для обменов. Полученные значения номеров относятся к группе `C_Group`.

Пример 6.2. Создание двух новых коммуникаторов расщеплением существующего

```
...
//Эти декларации должны быть глобальными!
int id, c_size, c_num; //информация о коммуникаторах
//Множество расчетных процессов разбито
//на l1 процессов по вертикали и l2 процессов по горизонтали
int l1, l2;
int left, right, upper, bottom; //номера соседей
MPI_Comm C_Group; //объявление нового коммуникатора
...
//Определение процессом своего номера в группе,
//связанной с коммуникатором MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD, &id);
...
// Создаем два новых коммуникатора
MPI_Comm_split(MPI_COMM_WORLD, id==0?0:1, 0, &C_Group); //(1)
MPI_Comm_size(C_Group, &c_size); //(2)
MPI_Comm_rank(C_Group, &c_num); //(3)
printf("My new comm_size=%d, rank=%d\n", c_size, c_num);
...
// узнаем соседей в новых коммуникаторах
if (c_num%l2==0) left=MPI_PROC_NULL;
else left=c_num-1; //(4)
if (c_num%l2==l2-1) right=MPI_PROC_NULL;
else right=c_num+1; //(5)
if (c_num/l2==0) upper=MPI_PROC_NULL;
else upper=c_num-l2; //(6)
if (c_num/l2==l1-1) bottom=MPI_PROC_NULL;
else bottom=c_num+l2; //(7)
...
MPI_Comm_free(C_Group);
...
```


В этом примере мы использовали фиктивный номер процесса `MPI_PROC_NULL` для тех случаев, когда соответствующий процесс-сосед отсутствует. Его семантика – отсутствующий узел. Фиктивный номер можно использовать при вызовах функций приема/передачи так же, как и обычный номер процесса: отличие только в том, что с данными при этом ничего не произойдет.

По окончании работы с группой не забудем вызвать `MPI_Comm_free(C_Group)` для освобождения коммуникатора.

Создание своих топологий. С точки зрения прикладного программиста создание виртуальной топологии связей аналогично перенумерации процессов согласно некоторому принципу. Например, в рамках нашего примера можно создать для всех вычислительных модулей декартову топологию, т. е. нумерацию в виде декартовых координат, в нашем случае – двумерных.

Решение с помощью виртуальных топологий состоит в следующем.

1. Создание из группы `C_Group` нового коммуникатора с ассоциированной с ним декартовой топологией (конструкторы `MPI_Cart_create()` или `MPI_Dims_create()`).

2. Перевод при необходимости номеров процессов в исходном коммуникаторе в его декартовы координаты и наоборот с помощью информационных функций `MPI_Cart_coords()` и `MPI_Cart_rank()` соответственно.

3. Определение единожды или каждый раз перед обменом данными абсолютных номеров (в новой группе) процессов-соседей с помощью вызова `MPI_Cart_shift()`.

Ниже кратко описаны упомянутые функции.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                   int *periods, int reorder, MPI_Comm *comm_cart);
```

Используя коммуникатор `comm_old`, связанный с некоторой группой процессов, функция в `comm_cart` возвращает новый коммуникатор, к которому подключается декартова топологическая информация. В `ndims` передается размерность создаваемой декартовой решетки, в массиве `dims` (размера `ndims`) задается количество процессов по каждой координате. Каждый элемент массива `periods` (размера `ndims`) определяет, обладает ли топология в данном направлении свойствами тора, т. е. существует ли связь между последним и первым элементами решетки в каждом направлении. Если `reorder = false`, то номер каждого процесса в новой группе идентичен номеру в старой группе. В противном случае функция может переупорядочивать процессы (возможно, чтобы обеспечить хорошее наложение виртуальной топологии на физическую систему). Если полная размерность

декартовой решетки меньше, чем размер группы коммуникаторов, то некоторые процессы возвращаются с результатом `MPI_COMM_NULL`. Вызов будет неверным, если он задает решетку большего размера, чем размер группы.

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int dims,  
                    int *coords);
```

Функция используется для перевода номера процесса `rank` в области связи, описываемой коммуникатором с декартовой топологией `comm`, в логические декартовы координаты процесса в этой области связи. Декартовы координаты возвращаются в массив `coords` размерности `dims`.

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank);
```

Для группы процессов с декартовой топологией, ассоциированной с коммуникатором `comm`, функция переводит логические координаты процессов `coords` в номера `rank`, которые используются в процедурах парного обмена. По сути, функция является обратной к `MPI_Cart_coords()`.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,  
                   int *rank_source, int *rank_dest);
```

Для группы процессов с декартовой топологией, ассоциированной с коммуникатором `comm`, функция используется для определения адресов процесса-отправителя `rank_source` и процесса-получателя `rank_dest` в парном обмене сообщениями при необходимости сдвига на `disp` позиций по решетке вдоль направления `direction`, т. е. аргумент `direction` указывает координату, по которой сдвигаются данные. Координаты маркируются от 0 до `ndims-1`, где `ndims` – число размерностей в топологии.

Функция учитывает признак периодичности декартовой топологии в указанном направлении координаты, возвращая номера процессов для кольцевого сдвига или сдвига без переноса. В случае сдвига без переноса в `rank_source` или `rank_dest` может быть возвращено значение `MPI_PROC_NULL` для указания, что процесс-отправитель или процесс-получатель при сдвиге вышли из диапазона.

Поскольку создание топологий ценно не само по себе, а с точки зрения оптимизации коммуникаций, то пример программы, использующей декартову топологию, приведен при обсуждении функции одновременного приема/передачи двухточечного обмена (пример 6.6).

С группой процессов в MPI возможно ассоциировать и наиболее общую топологию – граф.

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index,  
                     int *edges, int reorder, MPI_Comm *comm_graph);
```

Используя коммуникатор `comm_old`, связанный с некоторой группой процессов, функция в `comm_graph` возвращает новый коммуникатор, к которому подключается топологическая информация о графе. Параметр `nnodes` задает общее количество узлов графа (узлы маркируются от 0 до `nnodes-1`). Массив `index` размера `nnodes` задает степень каждой из вершин (число соседей), при этом `i`-й элемент массива хранит общее число соседей первых `i`-х вершин графа. Массив `edges` описывает ребра графа (размер массива равен общему количеству ребер графа). Если количество вершин графа `nnodes` меньше, чем размер группы коммуникатора, то некоторые процессы возвращают значение `MPI_COMM_NULL`. Вызов будет неверным, если он определяет граф большего размера, чем размер группы исходного коммуникатора.

Проиллюстрируем MPI-описание топологии графа для группы, состоящей из четырех процессов (табл. 6.1). Матрица смежности графа (`nnodes=4`) задана в первых двух строках табл. 6.1. Следующие две строки содержат описание массивов, являющихся входными для функции `MPI_Graph_create()`.

Таблица 6.1

Номер процесса (номер вершины графа)	0	1	2	3
Номера соседей	1,3	0	3	0,2
Значения элементов в массиве <code>index</code>	2	3	4	6
Значения элементов в массиве <code>edges</code>	1,3	0	3	0,2

Аналогом информационных функций о соседях в топологии графа являются функции `MPI_Graph_neighbors_count()` получения степеней вершин и функция `MPI_Graph_neighbors()` определения по линейному номеру процесса линейных номеров его соседей.

Отметим, что во многих случаях простое создание отдельных групп процессов предпочтительнее, чем использование топологий, поскольку отражает актуальное правило параллельного программирования: писать настолько просто, насколько возможно, до тех пор, пока это не вредит производительности.

Типы данных и предопределенные константы

Обсудим типы данных, с которыми могут оперировать MPI-вызовы. Тип пересылаемых данных задается не средствами языка программирования, а средствами MPI. Каждому базисному типу в языке программирования (в нашем случае в Си) соответствует предопределенный тип в MPI,

использующийся для указания типа пересылаемых элементов сообщений (табл. 6.2). Типы `MPI_BYTE` и `MPI_PACKED` не имеют аналогов в языках Си или ФОРТРАН. Значением типа `MPI_BYTE` является байт (8 двоичных цифр). Байт не интерпретируется и отличен от символа. Тип `MPI_PACKED` используется для пересылки упакованных данных и может описывать данные любого допустимого типа.

Таблица 6.2

Константы MPI	Тип в Си
<code>MPI_CHAR</code>	<code>signed char</code>
<code>MPI_SHORT</code>	<code>signed int</code>
<code>MPI_INT</code>	<code>signed int</code>
<code>MPI_LONG</code>	<code>signed long int</code>
<code>MPI_UNSIGNED_CHAR</code>	<code>unsigned char</code>
<code>MPI_UNSIGNED_SHORT</code>	<code>unsigned int</code>
<code>MPI_UNSIGNED</code>	<code>unsigned int</code>
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>
<code>MPI_FLOAT</code>	<code>float</code>
<code>MPI_DOUBLE</code>	<code>double</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>
<code>MPI_BYTE</code>	Нет соответствующего типа
<code>MPI_PACKED</code>	Нет соответствующего типа

В стандарте MPI существует несколько собственных предопределенных типов:

- `MPI_Status` — структура с полями, содержащими атрибуты `MPI_Source`, `MPI_Tag` и `MPI_Error` принимаемого сообщения;
- `MPI_Request` — системный тип, идентифицирующий неблокирующую операцию отправки/приема сообщения;
- `MPI_Comm` — системный тип, идентифицирующий коммуникатор.

В MPI предопределены также константы:

- `MPI_COMM_WORLD` — зарезервированный идентификатор группы, состоящей из всех процессов приложения;
- `MPI_ANY_SOURCE` — аргумент-джокер функций получения сообщения, означающий, что сообщение может быть принято «от кого угодно»;
- `MPI_ANY_TAG` — аргумент-джокер функций получения сообщения, означающий, что сообщение может быть принято с «каким угодно» тегом;
- `MPI_COMM_NULL` — константа-пустышка «неопределенный коммуникатор»;
- `MPI_PROC_NULL` — константа-пустышка «отсутствующий узел»;
- `MPI_DATATYPE_NULL` — константа-пустышка «неопределенный тип данных»;

- `MPI_REQUEST_NULL` — константа-пустышка «неопределенный идентификатор операции приема/передачи сообщения».

- `MPI_UNDEFINED` — константа неопределенного значения параметра `split`, используемая в процедуре `MPI_Comm_Split()`.

Практически все функции обмена предполагают, что данные однородны (одного типа) и непрерывно расположены в памяти. Логика приложения зачастую требует послышки разнородных данных (структур) или данных, непоследовательно расположенных в памяти (несмежные блоки массива). MPI предоставляет пользователю возможность на основе базовых типов конструировать типы более общей структуры. Подробно об определении пользовательских типов данных можно прочитать в стандарте MPI [79, 88] или монографиях [13, 14, 25–27, 32].

Для иллюстрации конструирования типов рассмотрим распространенный пример. Пусть имеется блок данных, обрабатываемый некоторым счетным узлом. Опишем блок как прямоугольный в общем случае массив, например, с именем `data[0...k1+1][0...k2+1]`, размером `k1+2` строк на `k2+2` столбцов. Периметр этого массива составляют «теневые грани», т. е. данные, принятые от соответствующих соседей (см., например, решение задачи Дирихле методом Якоби в гл. 5). Построим несколько производных типов для организации обменов с соседями и диспетчером.

Удобно объявить с помощью `#define` идентификатор для базового вещественного типа всей программы. Это может оказаться исключительно полезным при необходимости поменять тип данных во всей программе.

```
#define real long double
#define mpi_real MPI_LONG_DOUBLE
#define rlett "%Lf"
```

Здесь мы воспользовались предопределенным MPI-типом `MPI_LONG_DOUBLE`, соответствующим базовому типу языка Си `long double`. Затем объявим имена типов, которые будут сконструированы:

```
MPI_Datatype line, lineh, AllLines, AllData, row;
```

Для конструирования типа необходимо знать длину строки массива. Для этого получим указатели на элементы массива, отстоящие друг от друга ровно на строку:

```
//Указатели на переменную предопределенного типа
MPI_INT MPI_Aint pt0, pt1;
MPI_Address(&data[1][1], &pt0);
MPI_Address(&data[2][1], &pt1);
```

Теперь опишем тип данных с семантикой «строка длины `k2`». Данные этого типа мы будем отдавать (и принимать) нашим соседям сверху и снизу. Воспользуемся самым простым конструктором типов `MPI_Type_contiguous()`, позволяющим собирать непрерывную область из

повторяющихся типов (в нашем случае это базовый тип `MPI_LONG_DOUBLE`, переопределенный в `#define` как `mpi_real`).

```
//Конструирование нового типа line
MPI_Type_contiguous (k2, mpi_real, &line);
//Регистрация в системе нового типа line
MPI_Type_commit (&line);
```

На основе созданного типа сконструируем тип данных «массив данных без теневых граней». Такой массив можно отправлять диспетчеру при выгрузке данных на него. Для этого воспользуемся более общим конструктором типов `MPI_Type_hvector()`, который позволяет «размножить» копии исходного типа (в нашем случае `line`) в поля, являющиеся равноотстоящими друг от друга блоками. Количество блоков `k1` задается в первом аргументе конструктора, количество элементов исходного типа задается вторым аргументом, расстояние между блоками, как расстояние между началами двух соседних блоков в байтах, задается в третьем аргументе. В нашем случае – это расстояние между начальными элементами двух соседних строк массива `pt1-pt0`.

```
MPI_Type_hvector (k1, 1, pt1-pt0, line, &AllLines);
MPI_Type_commit (&AllLines);
```

Этим же конструктором воспользуемся для создания типа данных «столбец высоты `k1`» для обмена с соседями слева и справа. Обратите внимание, что исходным типом для конструктора на этот раз является тип элемента матрицы `mpi_real`:

```
MPI_Type_hvector (k1, 1, pt1-pt0, mpi_real, &row);
MPI_Type_commit (&row);
```

Наконец, повторим алгоритм для получения типа данных «массив вместе с теневыми гранями». Этот массив может присылать нам диспетчер при первой инициализации, чтобы на каждом вычислительном узле заполнить сразу и теневые грани, и тело массива:

```
//Конструирование вспомогательного типа
//«строка с теневыми элементами»
MPI_Type_contiguous (k2+2, mpi_real, &lineh);
MPI_Type_commit (&lineh);
//Конструирование из lineh развертки матрицы
//со всеми теневыми элементами
MPI_Type_hvector (k1+2, 1, pt1-pt0, lineh, &AllData);
MPI_Type_commit (&AllData);
```

Отметим, что в разбираемом примере MPI-программа выполняется `p+1` процессом, причем `p` процессов, предназначенных для вычислений, разбиты на `l1` групп по `l2` процессов в каждой. Все вызовы делаются на всех узлах – и на вычислительных, и на диспетчере. Разница только лишь

в том, что на диспетчере массив `data` имеет размер не $[k1+2][k2+2]$, а $[N+2][N+2]$, где N – это общая размерность задачи, $N = k1 \cdot l1 = k2 \cdot l2$.

По окончании работы с программой будет правильно освободить все созданные типы от необходимости существовать далее. Это делается с помощью вызова вида

```
MPI_Type_free(&line);
```

MPI располагает также следующими конструкторами типов: `MPI_Type_vector()`, `MPI_Type_struct()`, `MPI_Type_indexed()` и `MPI_Type_hindexed()`.

6.5. Организация взаимодействий процессов

Для любой функции MPI можно указать три основных признака.

Блокирующие функции останавливают (блокируют) выполнение процесса до завершения требуемой операции. *Неблокирующие функции* возвращают управление немедленно, выполнение операции продолжается в фоновом режиме. Таким образом, неблокирующая функция – это заявка на выполнение операции, чтобы проследить за удовлетворением системой заявки, одним из выходных параметров функции является квитанция – переменная специального типа `MPI_Request`. В MPI определен ряд функций проверки окончания неблокирующей операции, в которых квитанция является входным параметром. До окончания операции система запрещает использовать переменные и массивы, которые являлись аргументами неблокирующей функции.

Локальные функции не инициируют пересылок данных между ветвями. Например, поскольку копии системных данных хранятся в каждой ветви программы, то большинство информационных функций являются локальными. Естественно, функция передачи `MPI_Send()` локальной не является, но функция приема `MPI_Recv()` локальная, поскольку пассивно ждет поступления данных, ничего не пытаясь сообщить другим ветвям.

Коллективные функции должны быть вызваны всеми процессами коммутатора, который является одним из входящих параметров. Несоблюдение этого правила приводит к ошибкам выполнения. Помимо групповых функций обмена к коллективным относятся, например, функции создания коммуникатора.

Например, функция `MPI_Barrier(...)` является блокирующей, не-локальной, коллективной, а функция `MPI_Isend(...)` неблокирующей, нелокальной и неколлективной.

В языке Си все функции имеют целый тип и в случае успешного завершения возвращают `MPI_SUCCESS`, в противном случае – код ошибки.

Двухточечные обмены

Обмен сообщениями – это, в общем-то, основной предмет рассмотрения стандарта MPI. Есть две большие группы функций приема/передачи: функции типа «точка – точка», вовлекающие в процесс только два различных процесса (назовем их парными), и групповые функции, вовлекающие в обмен всю группу.

Парные функции, в свою очередь, можно разделить по двум характеристикам: режим работы и наличие блокировки. *Неблокирующей функцией* приема/передачи называют такую, которая всегда сначала возвращает управление в программу, а потом начинает заниматься передачей данных. Возврат из такой функции говорит лишь о том, что наша заявка на прием или передачу учтена системой и когда-нибудь будет исполнена, а мы в это время вольны заняться чем-либо еще. Для получения действительного статуса процесса передачи есть ряд функций `MPI_Wait*()` и `MPI_Test*()`. Таким образом, корректное программирование неблокирующей передачи обычно предполагает два действия: инициализацию передачи (приема) и проверку факта ее завершения. Для указания на неблокирующий характер функции к ней приписывают префикс «I», например: `MPI_Isend()` – неблокирующий вариант функции `MPI_Send()`.

Неблокирующая функция приема (`MPI_IRecv()`) сообщает системе, что можно начать (или ожидать) прием в фоновом режиме. Завершен ли этот прием, необходимо проверить дополнительно.

Блокирующая функция не возвращает управление, пока не выполнена какая-то существенная и вполне определенная часть приема или передачи. Какая именно часть обмена выполнится, зависит от режима работы вызванной функции.

Режимов существует четыре. *Синхронная передача* (`MPI_Ssend()`): возврат из функции передачи означает, что данные уже начали передаваться получателю непосредственно из того участка в памяти, где они лежат. Если это блокирующий вызов, то по возвращении из него известно, что буфер передачи уже можно использовать. *Буферизованная передача* (`MPI_Bsend()`): возврат из функции передачи происходит, когда передаваемое сообщение взято в системный или пользовательский буфер, при возврате управления из функции буфер передачи также готов к повторному использованию. *Передача по готовности* (`MPI_Rsend()`) похожа на синхронную передачу, но проверка готовности получателя принимать данные не выполняется, т. е. если в момент инициализации передачи получатель не готов, то происходит ошибка. Передача по готовности в некотором смысле реализует рандеву, т. е. передача состоится только в том случае, если получатель ее ожидает. И наконец, *автоматический режим* передачи (`MPI_Send()`) предполагает, что система сама делает выбор в пользу син-

хронной или буферизованной передачи на основании размера передаваемых данных.

Блокирующая функция приема существует одна для всех режимов (`MPI_Recv()`), и возврат из нее означает, что данные приняты.

Обобщенные сведения об основных функциях двухточечного обмена содержатся в табл. 6.3.

Таблица 6.3

Режим передачи	Блокирующая форма	Неблокирующая форма
Стандартный режим передачи	<code>MPI_Send(...)</code> ; (пример 6.3) Локальная операция. <i>Начинается</i> независимо от того, был ли зарегистрирован соответствующий прием. <i>Завершается</i> после освобождения системного буфера: сообщение «ушло в Сеть», но может еще некоторое время «гулять по Сети». Система выбирает реализацию самостоятельно – завершенность операции может означать, что сообщение буферизовано на стороне отправителя, а может означать, что инициировано начало приема, а сообщение сразу скопировано в буфер принимающего процесса	<code>MPI_Isend(...)</code> ; (пример 6.4) Локальная операция. <i>Инициализация передачи</i> означает формирование запроса на выполнение операции обмена и связывание его со специальным идентификатором операции (выходной параметр). <i>Успешность проверки</i> гарантирует, что сообщение ушло в Сеть
Синхронный режим передачи	<code>MPI_Ssend(...)</code> ; Нелокальная операция. <i>Начинается</i> независимо от того, был ли зарегистрирован соответствующий прием. <i>Завершается</i> при получении уведомления о начале приема процессом-получателем. Сообщение не буферизуется на стороне отправителя, по окончании операции буфер может использоваться дальше. Передача хотя и медленная, но не позволяет Сети переполняться «гуляющими» сообщениями	<code>MPI_Issend(...)</code> ; Локальная операция. <i>Инициализация передачи</i> означает формирование запроса на выполнение операции обмена и связывание его со специальным идентификатором операции (выходной параметр). <i>Успешность проверки</i> гарантирует, что начат прием сообщения
Буферизованный режим передачи	<code>MPI_Bsend(...)</code> ; Локальная операция. Перед использованием необходимо выделить буфер с помощью функции <code>MPI_Buffer_attach(...)</code> ;	<code>MPI_Ibsend(...)</code> ; (пример 6.5) Локальная операция. <i>Инициализация передачи</i> означает формирование запроса на выполнение операции обмена и связывание его со специальным идентификатором операции (выходной параметр).

Режим передачи	Блокирующая форма	Неблокирующая форма
	<p><i>Начинается</i> не зависимо от того, были зарегистрирован соответствующий прием.</p> <p><i>Завершается</i> сразу, поскольку сообщение копируется в выделенный буфер, в котором ожидает операции обмена. При завершении операции необходимо освободить буфер с помощью функции</p> <pre>MPI_Buffer_detach(...);</pre> <p>которая дожидается завершения выполнения обмена и обнуляет буфер</p>	<p><i>Успешность проверки</i> гарантирует, что отправляемые данные ушли в Сеть или скопированы в выделенный буфер. После этого передачу нельзя отменить. Если не будет зарегистрирован соответствующий прием, то буфер уже нельзя будет освободить.</p> <p>Для работы с буфером также следует использовать функции</p> <pre>MPI_Buffer_attach(...);</pre> <pre>MPI_Buffer_detach(...);</pre>
Передача по готовности	<pre>MPI_Rsend(...);</pre> <p>Нелокальная операция. <i>Выполняется только</i>, если инициирован соответствующий прием.</p> <p>Если прием не зарегистрирован, результат выполнения операции не определен. Сообщение просто выбрасывается в Сеть без буферизации.</p> <p><i>Завершается</i> сразу, не гарантирует получение сообщения.</p> <p>Самый быстрый (не используются этапы установки межпроцессорных связей), но и самый опасный режим передачи, при отладке рекомендуется заменять стандартным</p>	<pre>MPI_Irecv(...);</pre> <p>Локальная операция. <i>Инициализация передачи</i> означает разрешение начать прием, который происходит в фоновом режиме. Если принимающей стороной прием к моменту вызова функции не инициализирован, то результат выполнения операции не определен.</p> <p><i>Успешность проверки</i> гарантирует наличие сообщения в буфере приема принимающего процесса</p>
Прием сообщения	<pre>MPI_Recv(...);</pre> <p>Локальная операция. Если длина сообщения меньше буфера приема, то изменяется только часть буфера. Если длина сообщения больше буфера приема, то прием блокируется.</p> <p>Информацию о длине получаемого сообщения можно узнать с помощью функции</p> <pre>MPI_Get_count(...);</pre> <p>Для получения сообщения от произвольного источника определена специальная константа</p> <pre>MPI_ANY_SOURCE,</pre> <p>для получения сообщения с произвольным тегом — <code>MPI_ANY_TAG</code>.</p> <p>Окончание приема гарантирует наличие сообщения в буфере приема</p>	<pre>MPI_Irecv(...);</pre> <p>Локальная операция. <i>Инициализация передачи</i> означает разрешение начать прием, который происходит в фоновом режиме.</p> <p><i>Успешность проверки</i> гарантирует наличие сообщения в буфере приема и освобождение системного буфера.</p>

Важным моментом, связанным с автоматическим режимом, является то, что критический размер массива не определяется стандартом и может изменяться в зависимости от реализации. С этим может быть связано такое явление, как прекращение работы программы с ростом ее размера, когда синхронный режим неожиданно для пользователя сменяется буферизованным, а памяти системе не хватает. Поэтому при передаче крупных блоков данных лучше управлять буферизацией явно (`MPI_Bsend()/MPI_Ibsend()`), а при передаче действительно крупных – не использовать буферизацию (т. е. пользоваться `MPI_Ssend()/MPI_Issend()`).

Ниже приведен синтаксис использования парных функций обмена с привязкой к языку Си.

Блокирующие операции обмена

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
              int dest, int msgtag, MPI_Comm comm);
```

Функция осуществляет посылку в автоматическом режиме сообщения с идентификатором `msgtag`, состоящего из `count` элементов типа `datatype`, процессу с номером `dest`. Все элементы сообщения расположены подряд в буфере `buf`. Значение `count` может быть нулем. Тип передаваемых элементов `datatype` должен указываться с помощью его предопределенных констант. Процесс может передавать сообщение самому себе.

Следует еще раз отметить, что возврат из подпрограммы `MPI_Send()` не означает ни того, что сообщение уже передано процессу `dest`, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший `MPI_Send()`.

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype,
               int dest, int msgtag, MPI_Comm comm);
```

Функция осуществляет посылку сообщения в синхронном блокирующем режиме. Ее параметры совпадают с параметрами функции `MPI_Send()`.

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype,
               int dest, int msgtag, MPI_Comm comm);
```

Функция осуществляет буферизированную посылку сообщения в блокирующем режиме. Ее параметры совпадают с параметрами функции `MPI_Send()`. Перед выполнением функции необходимо определить и присоединить буфер достаточного размера функцией `MPI_Buffer_attach()`. После выхода из функции следует освободить этот буфер функцией `MPI_Buffer_attach()`. Отметим, что буфер должен быть описан как массив размером не менее `count+MPI_BSEND_OVERHEAD`, его не следует ис-

пользовать для других целей, например в качестве буфера в самой функции `MPI_Bsend()`.

```
int MPI_Buffer_attach(void* buffer, int size);
```

Функция выделяет буфер `buffer` в памяти прикладной программы, который нужно использовать для буферизации передаваемых сообщений в соответствующем режиме. Для вызова одной передачи можно присоединить только один буфер.

```
int MPI_Buffer_detach(void* buffer, int size);
```

Функция блокирует работу процесса до тех пор, пока все сообщения, находящиеся в буфере, не будут обработаны, а затем освобождает буфер `buffer`, но не память, им занимаемую.

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype,  
               int dest, int msgtag, MPI_Comm comm);
```

Функция осуществляет посылку сообщения в блокирующем режиме по готовности. Ее параметры совпадают с параметрами функции `MPI_Send()`.

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,  
              int source, int msgtag, MPI_Comm comm,  
              MPI_Status *status);
```

Процедура осуществляет прием сообщения с идентификатором `msgtag` от процесса `source` с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения `count`. Если число принятых элементов меньше значения `count`, то гарантируется, что в буфере `buf` изменятся только элементы, соответствующие принятому сообщению. Если нужно узнать точное число элементов в сообщении *до* или *после* его получения, можно воспользоваться функцией `MPI_Probe()`.

В качестве номера процесса-отправителя можно указать предопределенную константу `MPI_ANY_SOURCE` — «можно получить сообщение от любого процесса». В качестве идентификатора принимаемого сообщения можно указать константу `MPI_ANY_TAG` — «можно получить сообщение с любым идентификатором».

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову `MPI_Recv()`, то первым будет принято то сообщение, которое было отправлено раньше.

Пример 6.3 содержит код программы, которую следует запускать на двух процессорах. Она осуществляет обмен сообщениями в автоматическом режиме. Если система будет иметь достаточно ресурсов (что для такой простой программы ожидаемо), то обмены выполнятся без проблем,

поскольку сообщение первого дошедшего до оператора `MPI_Send()` процесса будет с большой долей вероятности буферизовано по инициативе системы и процесс перейдет к выполнению операции получения `MPI_Recv()`. Заметим, однако, что такой обмен не совсем безопасен. Если функции отправки и получения сообщения вызывать в двух различных циклах длиной, например, в 100000 итераций, то вполне вероятно истощение системных ресурсов и, как следствие, зависание программы. Более того, программа гарантированно зависнет, если операции отправки и приема сообщения поменять местами.

Получение информации о сообщении

Перед началом операции приема сообщения можно узнать о его наличии в буфере приема, а также о параметрах и даже структуре прибывшего сообщения с помощью функций `MPI_Get_Count`, `MPI_Probe()` и `MPI_Iprobe()`. Эти функции возвращают в структуру специального типа `MPI_Status` сведения об отправителе и теге, а также сообщают информацию о длине сообщения.

```
int MPI_Get_Count(MPI_Status *status, MPI_Datatype datatype,
                  int *count);
```

Функция определяет число уже принятых (если обращение к функции произошло после обращения к `MPI_Recv()`) или принимаемых (если обращение к функции произошло после обращения к `MPI_Probe()` или `MPI_Iprobe()`) элементов типа `datatype` сообщения со статусом `status`.

```
int MPI_Probe( int source, int msgtag, MPI_Comm comm,
               MPI_Status *status);
```

Функция обеспечивает получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение в области связи, ассоциированной с коммуникатором `comm`, с подходящим идентификатором `msgtag` и номером процесса-отправителя `source` не будет доступно для получения. Атрибуты доступного сообщения можно определить с помощью параметра `status`. Следует обратить внимание, что подпрограмма определяет только факт прихода сообщения, но реально его не принимает. В качестве идентификатора сообщения и номера процесса-отправителя могут быть указаны предопределенные константы `MPI_ANY_TAG` и `MPI_ANY_SOURCE` соответственно.

```
int MPI_Iprobe( int source, int msgtag, MPI_Comm comm,
               int *flag, MPI_Status *status);
```

Функция обеспечивает получение информации о структуре ожидаемого сообщения без блокировки. В параметр `flag` возвращается значение

1, если сообщение в области связи, ассоциированной с коммуникатором `comm`, с подходящим идентификатором `msgtag` и номером процесса-отправителя `source`, уже может быть принято (в этом случае действие функции полностью аналогично `MPI_Probe()`), и значение 0, если сообщения с указанными атрибутами еще нет.

Пример 6.3. Блокирующий обмен сообщениями в автоматическом режиме

```
#include <mpi.h>
#include <stdio.h>
#define leng 20 //длина передаваемого сообщения
int main( int argc, char **argv ){
    double tb,te;
    int i,rank,size;
    char WR[leng];
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    if (!rank){
        if (size!=2){
            printf("d: number of processes must be 2!\n", rank);
            MPI_Abort(MPI_COMM_WORLD,MPI_ERR_OTHER);
        }
    }
    sprintf(WR,"Hello from %d",rank); //формирование сообщения
    //Фиксация локального для процесса времени «начала обмена»
    tb=MPI_Wtime();
    MPI_Send(WR,leng,MPI_CHAR,size-
(rank+1),rank,MPI_COMM_WORLD);
    MPI_Recv(WR,leng,MPI_CHAR,size-(rank+1), size-(rank+1),
        MPI_COMM_WORLD,&status);
    //Фиксация локального для процесса времени «окончания обмена»
    te=MPI_Wtime();
    //Вывод сообщения и времени, затраченного на обмен
    printf("%d: WR=%s\n",rank,WR);
    printf("%d: Time=%le\n",rank,(te-tb));
    MPI_Finalize();
    return 0;
}
```

Поскольку сообщение может быть принято с аргументами-джокерами («принимай что угодно» – `MPI_ANY_TAG`, «от кого угодно» – `MPI_ANY_SOURCE`), то в структуре `status` сохраняются реально принятые сведения об источнике и сообщении. Поле структуры `status` `MPI_ERROR`, как правило, проверять необязательно – обработчик ошибок, устанавливаемый MPI по умолчанию, в случае сбоя завершит выполнение програм-

мы до возврата из `MPI_Recv()`. Отметим, что структура типа `MPI_Status` не содержит данные о фактической длине пришедшего сообщения. Длину сообщения следует узнавать с помощью информационной функции `MPI_Get_count()`.

Неблокирующие операции обмена

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
               int dest, int msgtag, MPI_Comm comm,
               MPI_Request *request);
```

Функция осуществляет посылку сообщения в неблокирующем режиме. Ее параметры совпадают с параметрами функции `MPI_Send()`. Последний параметр `request` является выходным и используется функциями семейств `MPI_Wait*()` и `MPI_Test*()` для проверки окончания выполнения операции.

```
int MPI_Issend(void *buf, int count, MPI_Datatype datatype,
                int dest, int msgtag, MPI_Comm comm,
                MPI_Request *request);
```

Функция осуществляет посылку сообщения в неблокирующем синхронном режиме. Ее параметры совпадают с параметрами функции `MPI_Isend()`.

```
int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype,
                int dest, int msgtag, MPI_Comm comm,
                MPI_Request *request);
```

Функция осуществляет буферизированную посылку сообщения в неблокирующем режиме. Ее параметры совпадают с параметрами функции `MPI_Isend()`. Перед выполнением функции необходимо определить и присоединить буфер достаточного размера функцией `MPI_Buffer_attach()`. После выхода из функции следует освободить этот буфер функцией `MPI_Buffer_detach()`.

```
int MPI_Irsend(void *buf, int count, MPI_Datatype datatype,
                int dest, int msgtag, MPI_Comm comm,
                MPI_Request *request);
```

Функция осуществляет посылку сообщения в неблокирующем режиме по готовности. Ее параметры совпадают с параметрами функции `MPI_Isend()`.

```
int MPI_IRecv(void *buf, int count, MPI_Datatype datatype,
               int source, int msgtag, MPI_Comm comm,
               MPI_Request *request);
```

Функция осуществляет прием сообщения аналогично `MPI_Recv()`, однако возврат из процедуры происходит сразу после инициализации про-

цесса приема без ожидания получения сообщения в буфере `buf`. Действительность получения сообщения можно проверить с помощью параметра `request` и функций семейств `MPI_Wait*()` и `MPI_Test*()`.

Сообщение, отправленное любой из процедур двухточечной послышки, может быть принято любой из функций `MPI_Recv()` и `MPI_IRecv()`.

Проверка выполнения обмена

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

Функция осуществляет ожидание завершения неблокирующих процедур отправки и получения сообщений, ассоциированных с идентификатором `request`. В случае приема атрибуты и длину полученного сообщения можно определить с помощью параметра `status`.

Существует еще несколько функций этого семейства. `MPI_Waitall()` ожидает завершения нескольких указанных в параметрах операций обмена. `MPI_Waitany()` ожидает завершения какой-либо одной из нескольких указанных в параметрах операций обмена, при этом если завершено несколько операций, то возвращается информация об одной случайно выбранной. `MPI_Waitsome()` ожидает завершения нескольких указанных в параметрах операций обмена, при этом если завершено несколько операций, то возвращается информация обо всех.

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status);
```

Функция осуществляет проверку завершения неблокирующих процедур отправки и получения сообщений, ассоциированных с идентификатором `request`. В параметр `flag` возвращается значение 1, если соответствующая операция завершена, и значение 0 – в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра `status`. Функция не блокирует процесс, она осуществляет только проверку.

Существует еще несколько функций этого семейства. `MPI_Testall()` осуществляет проверку завершения всех из указанных в параметрах операций обмена. `MPI_Testany()` осуществляет проверку завершения какой-либо одной из нескольких указанных в параметрах операций обмена; при этом если завершено несколько операций, то возвращается информация об одной случайно выбранной. `MPI_Testsome()` осуществляет проверку завершения нескольких указанных в параметрах операций обмена, при этом если завершено несколько операций, то возвращается информация обо всех.

Пример 6.4 содержит код программы, демонстрирующей обмен сообщениями в неблокирующем режиме. В отличие от примера 6.3 в каждом процессе прием сообщения стоит перед посылкой. Несмотря на это, программа благополучно завершится.

Пример 6.4. Неблокирующий обмен сообщениями в автоматическом режиме

```
#include <mpi.h>
#include <stdio.h>
#define leng 20 //размер буфера для сообщения WR-string
int main( int argc, char **argv ){
    double tb,te;
    int i,rank,size;
    char WR[leng],WR1[leng]; //буферы для сообщений
    //Структура, в которую записываются атрибуты сообщения
    MPI_Status status;
    //Квитанция для проверки статуса сообщения функцией
    MPI_Wait()
    MPI_Request req1,req2;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    if (!rank){
        if (size!=2){
            printf("%d: number of processes must be 2!\n",rank);
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }
    }
    sprintf(WR,"Hello from %d",rank); //формирование сообщения
    //Фиксация локального для процесса времени «начала обмена»
    tb=MPI_Wtime();
    MPI_Irecv(WR1, leng, MPI_CHAR, rank, size-rank+1),
        MPI_COMM_WORLD, &req2);
    MPI_Isend(WR, leng, MPI_CHAR, size-(rank+1), size-(rank+1),
        MPI_COMM_WORLD,&req1);
    MPI_Wait(&req1,&status);
    MPI_Wait(&req2,&status);
    //Фиксация локального для процесса времени «окончания обмена»
    te=MPI_Wtime();
    //Вывод сообщения и времени, затраченного на обмен
    printf("%d: WR=%s\n",rank,WR1);
    printf("%d: Time=%le\n",rank,(te-tb));
    MPI_Finalize();
    return 0;
}
```

Пример 6.5 демонстрирует корректный буферизованный обмен.

Пример 6.5. Неблокирующий обмен сообщениями в буферизованном режиме

```

#include <mpi.h>
#include <stdio.h>
#define BUFSIZE 2000 //размер буфера для буферизованного обмена
#define leng 20 //размер буфера для сообщения WR-string
int main( int argc, char **argv ){
    double t,t2;
    int i,zero,rank,size;
    char WR[leng],WR1[leng]; //буферы для сообщения
    char *buffer; //буфер для буферизованного обмена
    //Структура, в которую записываются атрибуты сообщения
    MPI_Status status;
    //Квитанция для проверки статуса сообщения функцией
    MPI_Wait()
    MPI_Request req1,req2;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    if (!rank){
        if (size!=2){
            printf("%d: number of processes must be 2!\n",rank);
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }
    }
    buffer=(char*)malloc(BUFSIZE);
    MPI_Buffer_attach(buffer,BUFSIZE); //выделение буфера
    sprintf(WR,"Hello from %d",rank);
    //Фиксация локального для процесса времени «начала обмена»
    tb=MPI_Wtime();
    MPI_IbSend(WR, leng, MPI_CHAR, size-(rank+1), rank,
               MPI_COMM_WORLD, &req1);
    MPI_Irecv(WR1, leng, MPI_CHAR, size-(rank+1), size-
              (rank+1),
               MPI_COMM_WORLD, &req2);
    MPI_Wait(&req1,&status);
    MPI_Wait(&req2,&status);
    //Фиксация локального для процесса времени «окончания обмена»
    te=MPI_Wtime();
    //Вывод сообщения и времени, затраченного на обмен
    printf("%d: WR=%s\n",rank,WR1);
    printf("%d: Time=%le\n",rank,(te-tb));
    MPI_Buffer_detach(&buffer,&zero); //освобождение буфера
    MPI_Finalize();
    return 0;
}

```

Использовать ли блокирующие или неблокирующие функции? Однозначного ответа нет. Если позволяет время, отведенное на разработку, лучше попробовать оба варианта. Если время пересылки данных много меньше времени счета, то использование неблокирующих функций только усложнит отладку, но ничего не даст в плане производительности; если же время пересылки сопоставимо со временем счета, то выигрыш от отсутствия блокировок может быть существенным.

Чего совершенно точно не стоит делать – это пересылать данные малыми порциями. Какова бы ни была коммуникационная сеть, производительность множества мелких передач всегда будет намного ниже, чем одной крупной.

Следовать ли стандарту в той части, которая запрещает доступ даже на чтение в буфер передачи при использовании неблокирующей небуферизующей передачи? Авторам не известны примеры вычислительных систем, для которых это требование имеет практический смысл; с другой стороны, это не значит, что их нет или не появится в обозримом будущем. Видимо, в программах, написанных с расчетом на длительное использование, на уровне архитектуры следует предусмотреть возможность соблюдения этого требования.

Отложенные обмены

Для снижения накладных расходов, возникающих в рамках одного процесса при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером, возможно объединение нескольких запросов на прием и/или передачу и дальнейшее их обслуживание одной командой. Способ приема сообщения никак не зависит от способа его отправки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

```
int MPI_Send_Init(void *buf, int count, MPI_Datatype datatype,
                  int dest, int msgtag, MPI_Comm comm,
                  MPI_Request *request);
```

Функция обеспечивает формирование запроса на выполнение пересылки данных. Все параметры точно такие же, как и у функции `MPI_Isend()`, однако в отличие от нее пересылка не начинается до вызова функции `MPI_Startall()`.

```
int MPI_Recv_Init(void *buf, int count, MPI_Datatype datatype,
                  int source, int msgtag, MPI_Comm comm,
                  MPI_Request *request);
```

Функция обеспечивает формирование запроса на выполнение приема данных. Все параметры точно такие же, как и у функции `MPI_Ireceive()`,

однако в отличие от нее реальный прием не начинается до вызова функции `MPI_Startall()`.

```
int MPI_Start_All (int count, MPI_Request *requests);
```

Функция обеспечивает запуск всех отложенных взаимодействий, ассоциированных с вызовами функций `MPI_Send_Init()` или `MPI_Recv_Init()` с элементами массива запросов `requests`. Все взаимодействия запускаются в режиме без блокировки, а их завершение можно определить с помощью функций `MPI_Wait()` или `MPI_Test()`.

Совмещение операций отправки и передачи

Существует также функция совмещенного приема и передачи `MPI_Sendrecv()`, которую рекомендуется по возможности применять:

```
int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype,  
                  int dest, int stag, void *rbuf, int rcount,  
                  MPI_Datatype rtype, int source, MPI_Datatype rtag,  
                  MPI_Comm comm, MPI_Status *status);
```

Функция совмещает в едином запросе отсылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией `MPI_Sendrecv()`, может быть принято обычным образом, и точно также операция `MPI_Sendrecv()` может принять сообщение, отправленное обычной операцией. Буферы приема и отправки обязательно должны быть различными. Функция имеет следующие параметры: `sbuf` – адрес начала буфера отправки сообщения; `scount` – число передаваемых в сообщении элементов; `stype` – тип передаваемых элементов; `dest` – номер процесса-получателя; `stag` – идентификатор посылаемого сообщения; `rbuf` – адрес начала буфера приема сообщения (выходной параметр); `rcount` – число принимаемых элементов сообщения; `rtype` – тип принимаемых элементов; `source` – номер процесса-отправителя; `rtag` – идентификатор принимаемого сообщения; `comm` – коммунитор; `status` – параметры принятого сообщения (выходной параметр).

Часто совмещение отправки и приема сообщений используется для сдвига по цепочке (кольцу) процессов. На рис. 6.1 отображено выполнение блокирующей совмещенной операции для восьми процессов. Из рисунка видно, что для того, чтобы цепочка блокирующих отправок и приемов сообщений сработала, необходима некоторая задержка. Самая большая задержка будет у инициатора отправки.

Как уже отмечалось при обсуждении топологий, использование отправки и передачи сообщений совместно с функцией определения номеров

процессов `MPI_Cart_shift()` реализует быстрое смещение данных вдоль какого-либо направления декартовой решетки.

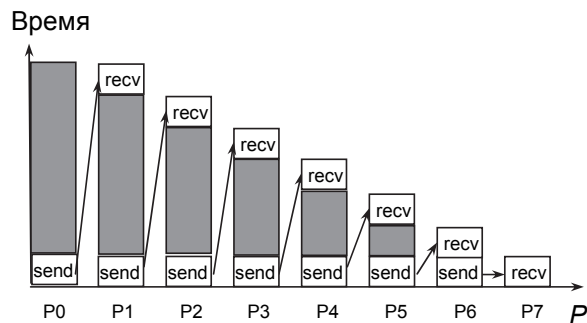


Рис. 6.1. Совмещение операций отправки и передачи для восьми процессов

1	1	1	1	1	1
1	2	3	4	5	6
1	3	6	10	15	21
1	4	10	20	35	56
1	5	15	35	70	126
1	6	21	56	126	252

Рис. 6.2. Фрагмент треугольника Паскаля (6×6)

Пример 6.6 демонстрирует использование такого приема. В этом примере производится итерационное построение прямоугольного фрагмента треугольника Паскаля. Напомним, что каждый элемент треугольника Паскаля равен сумме элементов, расположенных на один элемент выше и на один элемент левее текущего (рис. 6.2). Построение начинается с единицы в верхнем левом углу. Предполагается, что процессы связаны в топологию непериодической 2D-решетки размерностью $N \times N$. Каждый процесс изначально хранит один элемент будущего треугольника (только верхний правый из них сначала отличен от нуля).

Пример 6.6. Построение фрагмента треугольника Паскаля

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    MPI_Comm grid_comm; //коммуникатор с топологией 2D-решетки
    MPI_Status status;
    int dims[2], periodic[2], coordinates[2]; //параметры топологии
    int N=10, reorder=1, ndims=2, maxdims=2; //описание топологии
    int size, rank, my_grid_rank;
    int source_right, source_down, dest_right,
    dest_down; //соседи
    int buf_out, buf_in=0;
    int i, all, right, down;
    dims[0]=dims[1]=N;
    periodic[0]=periodic[1]=0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size > N*N) MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
```


Пример 6.6 (окончание). Построение фрагмента треугольника Паскаля

```

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) all=1; else all=0;
right=0; down=0;
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periodic,
                reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank, maxdims,
                coordinates);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Cart_shift(grid_comm, 1, 1, &source_right, &dest_right);
MPI_Cart_shift(grid_comm, 0, 1, &source_down, &dest_down);
for (i=0; i<2*N-2; i++){
    buf_out=all-right;
    MPI_Sendrecv(&buf_out, 1, MPI_INT, dest_right, i, &buf_in, 1,
                MPI_INT, source_right, i, grid_comm, &status);
    right=all; all=all+buf_in; buf_in=0;
    buf_out=all-down;
    MPI_Sendrecv(&buf_out, 1, MPI_INT, dest_down, i, &buf_in, 1,
                MPI_INT, source_down, i, grid_comm, &status);
    down=all; all=all+buf_in; buf_in=0;
    MPI_Barrier(MPI_COMM_WORLD)
}
printf("process (%i,%i) has %i\n",
        coordinates[0], coordinates[1], all);
MPI_Finalize();
return 0;
}

```

Алгоритм состоит в следующем. Каждый процесс имеет счетчики: `all` – количество единиц, переданных элементу всего; `right` – количество единиц, посланных элементом направо; `down` – количество единиц, посланных элементом вниз. В начале вычислений один раз с помощью функции `MPI_Cart_shift()` определяются номера соседей для совместных операций отправки/приема со сдвигом по решетке процессов вправо (`source_right`, `dest_right`) и со сдвигом по решетке процессов вниз (`source_down`, `dest_down`).

Далее в цикле $2 \cdot N - 2$ раза процессы обмениваются значениями, причем распространение информации идет вправо и вниз. Каждый обмен сопровождается пересчетом счетчиков. На рис. 6.2 отображен результат работы программы на решетке из 6×6 процессов.

Упаковка данных

Практически все функции обмена предполагают, что данные однородны (одного типа) и непрерывно расположены в памяти. Для организа-

ции пересылки неоднородных данных выше был описан механизм создания производных типов. В MPI предусмотрен и явный способ сборки и разборки сообщений, содержащих значения разных типов и располагаемых в разных областях памяти.

Для использования данного подхода должен быть определен буфер памяти достаточного размера для сборки сообщения. Входящие в состав сообщения данные упаковываются в буфер, пересылаются и распаковываются на стороне адресата.

```
int MPI_Pack (void* inbuf, int incount, MPI_Datatype datatype,
               void *outbuf, int outsize, int *position,
               MPI_Comm comm);
```

Функция пакетует сообщение в буфер послыки, описанный аргументами `inbuf` (начало входного буфера), `incount` (число единиц входных данных), `datatype` (тип данных каждой входной единицы) в буферном пространстве. Входным буфером может быть любой коммуникационный буфер, разрешенный в `MPI_SEND`. Выходной буфер есть смежная область памяти, содержащая `outsize` байтов, начиная с адреса `outbuf` (длина подсчитывается в байтах, а не в элементах, как если бы это был коммуникационный буфер для сообщения типа `MPI_PACKED`).

Параметр `position` определяет первую ячейку в выходном буфере, которая должна быть использована для упаковки. Функция увеличивает `position` на размер упакованного сообщения, и выходное значение `position` есть первая ячейка в выходном буфере, следующая за ячейками, занятыми упакованным сообщением. Таким образом, начальное значение переменной `position` должно быть сформировано до начала упаковки, а далее оно переустанавливается функцией `MPI_Pack()`. В аргументе `comm` передается коммуникатор, который будет использован для передачи упакованного сообщения.

Для определения необходимого размера буфера для упаковки может быть использована функция

```
int MPI_Pack_size (int count, MPI_Datatype type,
                   MPI_Comm comm, int *size);
```

Функция определяет размер `size`, необходимый для буфера, в который следует упаковать `count` элементов типа `type`.

После упаковки всех необходимых данных подготовленный буфер может быть использован в функциях передачи данных с указанием типа `MPI_PACKED`.

```
int MPI_Unpack (void* inbuf, int insize, int *position,
                 void *outbuf, int outcount, MPI_Datatype datatype,
                 MPI_Comm comm)
```

Функция распаковывает сообщение в приемный буфер, описанный аргументами `outbuf` (начало входного буфера), `outcount` (число элементов типа `datatype` для распаковки) из буферного пространства. Выходным буфером может быть любой коммуникационный буфер, разрешенный в `MPI_RECV`. Входной буфер есть смежная область памяти, содержащая `insize` байтов, начиная с адреса `inbuf`.

Пример 6.7. Обмен с упаковкой данных

```
#include <string.h>
#include <mpi.h>
#include <stdio.h>
#define BUFSIZE 100
#define BUF 20
int main(int argc, char* argv[]){
    int i,j,count=100; //количество записей для каждого процесса
    int rank, size, position=0;
    struct element{ //структура записи
        int number;
        char fio[BUF];
        float height;
    };
    struct element *klass;
    //Имена выходного и входного файлов
    char fileO[24]="Outfile.dat", fileI[24];
    FILE *f_out,*f_in;
    char buffer[BUFSIZE], buffer2[BUFSIZE];
    MPI_Status status;
    int beanpole; //номер самого высокого
    char s[BUF]; //фамилия самого высокого
    float hght; //рост самого высокого
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    //Выделение памяти
    klass=(struct element *)calloc(count+1,
                                   sizeof(struct element));

    //Имя входного файла зависит от номера процесса
    sprintf(fileI,"Infile_%d.dat",rank);
    if ((f_in = fopen(fileI, "r")) == NULL)
        printf ("\n Cannot open %s file.\n",fileI);
    //Чтение данных
    for (i=0;i<count;i++){
        fscanf(f_in,"%d",&klass[i].number);
        fscanf(f_in,"%s",&klass[i].fio);
        fscanf(f_in,"%f",&klass[i].height);
    }
    //определение фамилии самого высокого в своей группе
    beanpole=0;
    for (i=1;i<count;i++)
        if (klass[i].height > klass[beanpole].height) beanpole=i;
```

Пример 6.7 (окончание). Обмен с упаковкой данных

```

//Упаковка фамилии и роста самого высокого в своей группе
//и отправка ее управляющему процессу
if (rank!=0){
    strcpy(s,klass[beanpole].fio);
    hght= klass[beanpole].height;
    MPI_Pack(&s, BUF, MPI_CHAR, buffer, BUFSIZE, &position,
            MPI_COMM_WORLD);
    MPI_Pack(&hght, 1, MPI_FLOAT, buffer, BUFSIZE, &position,
            MPI_COMM_WORLD);
    MPI_Send(buffer,BUFSIZE,MPI_PACKED,0,
            rank,MPI_COMM_WORLD);
}
else {
    //Управляющий процесс печатает свой результат,
    //затем принимает сообщения, распаковывает и печатает
    if ((f_out = fopen(file0, "w")) == NULL)
        printf ("\n Cannot open %s file.\n",file0);
    fprintf(f_out,"%d: %s with height %f \n",
            rank,klass[beanpole].fio, klass[beanpole].height);
    for (i=1;i<size;i++) {
        MPI_Recv(buffer2, BUFSIZE, MPI_PACKED, i, i,
                MPI_COMM_WORLD, &status);
        MPI_Unpack(buffer2, BUFSIZE, &position, &s, BUF,
                MPI_CHAR, MPI_COMM_WORLD);
        MPI_Unpack(buffer2, BUFSIZE, &position, &hght, 1,
                MPI_FLOAT, MPI_COMM_WORLD);
        fprintf(f_out,"%d: %s with height %f\n",i,s,hght);
    }
    fclose(f_out);
}
fclose(f_in);
MPI_Finalize();
return 0;
}

```

Параметр `position` определяет первую ячейку во входном буфере, которая должна быть использована для упаковки. Функция увеличивает `position` на размер упакованного сообщения, и выходное значение `position` есть первая ячейка в выходном буфере, следующая за ячейками, занятыми упакованным сообщением. Таким образом, начальное значение переменной `position` должно быть сформировано до начала распаковки и далее переустанавливается функцией `MPI_Unpack()`. В аргументе `comm`

передается коммунитор, который был использован для передачи упакованного сообщения.

Вызов функции `MPI_Unpack()` осуществляется последовательно для распаковки всех упакованных данных, при этом порядок распаковки должен соответствовать порядку упаковки.

Поскольку упаковка данных является дополнительным действием, то ее нужно применять с осторожностью. Упаковка оправдана при сравнительно небольших размерах сообщений и при малом количестве повторений или при явном использовании буферов для буферизованного режима передачи данных. В других случаях удобнее конструировать типы, поскольку операция создания и регистрации типа выполняется один раз.

Упаковка данных продемонстрирована на примере определения самого высокого человека в группе (пример 6.7). Каждый процесс считывает данные о группе учащихся из своего файла. Данные хранятся по записям. Процесс находит самого высокого участника своей группы и сообщает об этом управляющему процессу (который, впрочем, также обрабатывает свой файл). Данные, которые пересылаются, разнородны – фамилия (строка символов) и рост (`float`), поэтому для удобства пересылки они упаковываются.

Коллективные обмены

Групповые функции необходимы для того, чтобы собрать с одного множества процессов некоторые данные, выполнить над ними какую-то операцию (предопределенную, определенную пользователем или не выполнять никакой) и разместить результат в другое множество процессов. Множества процессов могут пересекаться или нет, состоять из одного или более процессов. Однако их объединение обязательно должно включать в себя все процессы некоторой группы. Прежде чем на всех процессах этой группы не будет вызвана одна и та же групповая функция (конечно, с разными параметрами), ни из одного вызова соответствующей функции управление не вернется. В этом смысле все групповые функции – *синхронные* и *блокирующие*.

Следует понимать, что смешивать парные и групповые функции нельзя, сообщение, посланное групповой функцией, не получится принять с помощью `MPI_Recv`, и наоборот.

Основные особенности, связанные с использованием функций коллективного взаимодействия процессов, обсуждались в гл. 5 (см., в частности, рис. 5.12, 5.13). Ниже приведен синтаксис использования этих функций с привязкой к языку Си.

Синхронизация барьером

```
int MPI_Barrier(MPI_Comm comm);
```

Функция синхронизации процессов блокирует работу процессов, вызвавших данную функцию, до тех пор, пока все оставшиеся процессы группы, ассоциированной с коммуникатором `comm`, также не вызовут эту функцию.

Глобальные функции связи

```
int MPI_Bcast (void *buf, int count, MPI_Datatype datatype,
               int source, MPI_Comm comm);
```

Функция осуществляет рассылку сообщения, содержащегося в `buf` с элементами `count` типа `datatype`, от процесса `source` всем процессам, включая рассылающий процесс. При возврате из функции содержимое буфера `buf` процесса `source` будет скопировано в локальный буфер процесса. Значения параметров `count`, `datatype` и `source` должны быть одинаковыми у всех процессов.

```
int MPI_Gather (void *sbuf, int scount, MPI_Datatype stype,
                void *rbuf, int rcount, MPI_Datatype rtype,
                int dest, MPI_Comm comm);
```

Функция осуществляет сбор данных со всех процессов, ассоциированных с коммуникатором `comm`, в буфере `rbuf` процесса `dest`. Каждый процесс, включая `dest`, посылает содержимое своего буфера `sbuf` процессу `dest`. Собирающий процесс сохраняет данные в буфере `rbuf`, располагая их в порядке возрастания номеров процессов. Параметр `rbuf` имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров `scount`, `rcount`, `stype`, `rtype` и `dest` должны быть одинаковыми у всех процессов. Существует расширяющая функциональные возможности функция `MPI_Gatherv()`, позволяющая использовать различное количество данных из каждого процесса, поскольку параметр `rcount` является массивом.

```
int MPI_Scatter (void* sbuf, int scount, MPI_Datatype stype,
                 void* rbuf, int rcount, MPI_Datatype rtype,
                 int root, MPI_Comm comm);
```

Функция является обратной к `MPI_Gather()`. Результат выглядит так, как будто корень выполнил n посылающих операций `MPI_Send()` всем процессам, ассоциируемым с коммуникатором `comm`, включая самого себя, а каждый принимающий процесс выполнил функцию `MPI_Recv()` от корневого процесса. Аргументы `scount` и `stype` в корне должны быть равны аргументам `rcount` и `rtype` во всех процессах. Это подразумевает, что количество посланных данных должно быть равно количеству полученных

данных, попарно между каждым процессом и корнем. Все аргументы в функции значимы на корневом процессе, в то время как на других процессах значимы только аргументы `rbuf`, `rcount`, `rtype`, `root`, `comm`. Посылающийся буфер игнорируется для всех процессов, не принадлежащих корню. Существует расширяющая функциональные возможности функция `MPI_Scatterv()`, позволяющая посылать различное количество данных каждому процессу, поскольку параметр `scount` является массивом.

```
int MPI_Allgather(void* sbuf, int scount, MPI_Datatype stype,  
                  void* rbuf, int rcount, MPI_Datatype rtype,  
                  MPI_Comm comm);
```

Функция аналогична `MPI_Gather()`, за исключением того, что все процессы, ассоциированные с коммуникатором `comm`, получают результат от всех процессов, а не только от одного корня, т. е. j -й блок данных, посланных из каждого процесса, получен каждым процессом и размещается в j -м блоке буфера `rbuf`. Аргументы `scount` и `stype` должны быть равны во всех процессах. Результат выполнения функции выглядит так, как будто все процессы выполнили n запросов к `MPI_Gather()` для `root = 0, \dots, n-1`. Правила использования `MPI_Allgather()` аналогичны соответствующим правилам для `MPI_Gather()`. Существует расширяющая функциональные возможности функция `MPI_Allgatherv()`, в которой позволяет использовать буферы различного размера.

```
int MPI_Alltoall(void* sbuf, int scount, MPI_Datatype stype,  
                 void* rbuf, int rcount, MPI_Datatype rtype,  
                 MPI_Comm comm);
```

Функция расширяет возможности `MPI_Allgather()` для случая, когда каждый процесс, ассоциированный с коммуникатором `comm`, посылает различные данные на каждый из приемников, j -й блок, посланный из процесса i , получен процессом j и размещен в i -м блоке буфера `rbuf`. Аргументы `scount` и `stype` в процессе должны быть равны аргументам `rcount` и `rtype` в любом другом процессе. Это подразумевает, что количество посланных данных должно быть равно количеству полученных попарно между каждой парой процессов. Все аргументы во всех процессах значимы. Существует расширяющая функциональные возможности функция `MPI_Alltoallv()`.

Глобальные функции приведения

```
int MPI_Allreduce(void *sbuf, void *rbuf, int count,  
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Функция осуществляет выполнение `count` глобальных операций `op` над значениями буфера `sbuf` с возвратом в буфер `rbuf` каждого процесса, ассоциированного с коммуникатором `comm`, `count` результатов. Операция

выполняется независимо над соответствующими аргументами всех процессов. Значения параметров `count` и `datatype` у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция `op` обладает свойствами ассоциативности и коммутативности. Список допустимых операций, предопределенных для глобальных функций приведения, представлен в табл. 6.4.

Таблица 6.4

Имена операций	Значения операций
<code>MPI_MAX</code>	Максимум
<code>MPI_MIN</code>	Минимум
<code>MPI_SUM</code>	Сумма
<code>MPI_PROD</code>	Произведение
<code>MPI_BAND</code>	Логическое «И»
<code>MPI_LOR</code>	Поразрядное «И»
<code>MPI_BOR</code>	Логическое «ИЛИ»
<code>MPI_XOR</code>	Поразрядное «ИЛИ»
<code>MPI_BXOR</code>	Логический «XOR»
<code>MPI_MAXLOC</code>	Поразрядный «XOR»
<code>MPI_MINLOC</code>	Максимальное значение и локализация
	Значение минимума и локализация

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm);
```

Функция аналогична предыдущей, но результат будет записан в буфер `rbuf` только у процесса `root`.

```
int MPI_Reduce_scatter(void* sbuf, void* rbuf, int *rcounts,
                      MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Функция предполагает сначала выполнение поэлементной редукции `op` на векторе из $\sum_i \text{rcounts}[i]$ элементов типа `datatype` в буфере отправки `sendbuf`. Далее полученный вектор результатов разделяется на `n` непересекающихся сегментов, где `n` – число процессов в группе. Сегмент `i` содержит `rcount[i]` элементов типа `datatype`, при этом он принимается в буфер `rbuf` `i`-го процесса.

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Функция выполняет префиксную редукцию данных, распределенных в группе. Операция возвращает в приемный буфер процесса `i` редукцию значений в посылающих буферах процессов с номерами `0, ..., i` (включительно). Тип поддерживаемых операций, их семантика и ограничения на буферы отправки и приема такие же, как и для `MPI_Reduce()`.

Пример 6.8. Создание новых групп для коллективных обменов

```

#include <mpi.h>
#include <stdio.h>
main(int argc, char **argv) {
    int me_ch, me_nch;
    int buf, error;
    int ranks_ch[] = {0, 2, 4, 6}, ranks_nch[] = {1, 3, 5, 7};
    int rank, size, sum;
    MPI_Group MPI_GROUP_WORLD, subgroup_ch, subgroup_nch;
    MPI_Comm the_comm_ch, the_comm_nch;
    MPI_Status status;
    MPI_Request req[3];
    MPI_Comm myComm; //интракоммуникатор для локальной под-
    группы
    int membershipKey;
    int intra_rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (!rank){
        if (size!=8){
            printf("%d: number of processes must be 8!\n", rank);
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }
    }
    //Создание группы процессов
    //на основе существующего коммуникатора
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
    // Разделение группы процессов на две подгруппы
    MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks_ch, &subgroup_ch);
    MPI_Group_rank(subgroup_ch, &me_ch);
    MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks_nch, &subgroup_nch);
    MPI_Group_rank(subgroup_nch, &me_nch);
    // Создание коммуникаторов на основе групп процессов
    MPI_Comm_create(MPI_COMM_WORLD, subgroup_ch, &the_comm_ch);
    MPI_Comm_create(MPI_COMM_WORLD, subgroup_nch, &the_comm_nch);
    // Коллективное взаимодействие внутри групп
    if (me_chet != MPI_UNDEFINED){
        MPI_Allreduce(&rank, &sum, 1, MPI_INT, MPI_SUM, the_comm_ch);
    }
    if (me_nchet != MPI_UNDEFINED){
        MPI_Allreduce(&rank, &sum, 1, MPI_INT, MPI_MIN, the_comm_nch);
    }
    printf("Rank=%d, value=%d\n", rank, sum);
    MPI_Finalize();
    return 0;
}

```

Отметим еще раз, что в процессе коллективного взаимодействия участвуют все процессы приложения, т. е. соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из функции коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или отправки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

В примере 6.8 на основе предопределенного коммуникатора создается группа, в которую входят все запущенные процессы (в примере их должно быть восемь). Затем группа расщепляется (`MPI_Group_incl()`) на две – в одну входят процессы с номерами, определяемыми в массиве `ranks_chet[] = {0, 2, 4, 6}`, а во вторую – в массиве `ranks_nechet[] = {1, 3, 5, 7}`. Процессы, не входящие в соответствующие группы, определяют свой ранг в ней как `MPI_UNDEFINED`. На основе каждой из групп создаются изолированные области связи, ассоциированные с интракоммуникаторами. Каждый процесс в первой группе с помощью коллективного обмена узнает сумму всех рангов этой группы, а во второй – минимальный среди всех рангов группы.

Умножение матрицы на вектор

Для демонстрации работы коллективных обменов приведем код программы, реализующий алгоритм умножения прямоугольной матрицы на вектор.

Размерности матрицы и вектора, как и число процессов, заранее не известны. Данные разрезаются на `size` линий, где `size` – количество процессов. Количество строк в линии зависит от количества строк в матрице, которое, в свою очередь, не обязательно кратно `size`. Выделение всей памяти происходит динамически. Поскольку распределение строк по процессам неравномерно, используются векторные аналоги операций приведения `MPI_Scatterv()` и `MPI_Gatherv()`.

6.6. Повышение эффективности MPI-программ

Эффективность программирования для кластерных систем сильно зависит от характеристик межпроцессорных связей. Для улучшения латентности и пропускной способности сети необходимо:

- применять архивацию данных для больших сообщений;

- использовать специальные типы данных вместо `MPI_PACK()` и `MPI_UNPACK()` там, где это возможно;
- для уменьшения количества функций обмена заменять там, где это возможно, `MPI_Send()` и `MPI_Recv()` на коллективные операции с созданием соответствующих коммунитаторов;
- поскольку применение `MPI_ANY_SOURCE` может увеличивать латентность, следует стараться точно определять номер принимающего процесса при вызове соответствующих функций MPI;
- в случаях, когда запросы на прием сообщений не могут быть выполнены сразу, предпочтительнее использовать `MPI_RECV_INIT()` и `MPI_STARTALL()` вместо вызова `MPI_Irecv()` в цикле.

Пример 6.9. Умножение матрицы на вектор

```
#include<mpi.h>
#include <stdio.h>
#include<stdlib.h>
int main(int argc, char** argv){
    char file0[10]="in.txt"; //файл с исходными данными
    FILE *f_in;
    int i,j, C1,C2,rows,cols;//количество строк и столбцов матриц
    int rank,size;
    int *matrix,*vector,*result,*matrix1,*rez;
    //Количество элементов, отсылаемых каждому процессу
    int *counts,*countsrez;
    //Смещение относительно начала массива
    //для функций MPI_Scatterv() и MPI_Gatherv()
    int *disp,*disprez;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    // данные считывает 0-й процессор
    if(!rank){
        if ((f_in = fopen(file0, "r")) == NULL)
            {printf ("\n Cannot open %s file.\n",file0); return 0;}
        fscanf(f_in,"%d",&rows); //считываем количество строк
        fscanf(f_in,"%d",&cols); //считываем количество столбцов
        //Выделяем память
        matrix=(int*)malloc(sizeof(int)*cols*rows);
        vector=(int*)malloc(sizeof(int)*cols);
        result=(int*)malloc(sizeof(int)*rows);
        for(i=0;i<cols*rows;i++)
            fscanf(f_in,"%d",matrix+i); //считываем матрицу из файла
        for(i=0;i<cols;i++)
            fscanf(f_in,"%d",vector+i); //считываем вектор из файла
    }
}
```

Пример 6.9 (продолжение). Умножение матрицы на вектор

```

//Рассылаем rows, cols и vector всем процессам
MPI_Bcast(&rows,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&cols,1,MPI_INT,0,MPI_COMM_WORLD);
//Прежде чем получить вектор все, кроме 0-го процесса,
//должны выделить память для него
if (rank) vector=(int*)malloc(sizeof(int)*cols);
MPI_Bcast(vector,cols,MPI_INT,0,MPI_COMM_WORLD);
//Выделяем память для вычислений
counts=(int*)malloc(sizeof(int)*size);
disp=(int*)malloc(sizeof(int)*size);
countsrez=(int*)malloc(sizeof(int)*size);
disprez=(int*)malloc(sizeof(int)*size);
matrix1=(int*)malloc(sizeof(int)*cols*rows);
rez=(int*)malloc(sizeof(int)*rows);
//вычисляем C1 и C2
C1=rows/size; C2=rows%size;
//Данные разрезаются на size линий.
//Первые size-C2 линий имеют C1 строк,
//а остальные C2 линий имеют C1+1 строку
//Для <size-C2
for (i=0;i<size-C2;i++){
    //количество элементов, посылаемых нулевым процессом i-му,
    counts[i]=C1*cols;
    //количество элементов, посылаемых i-м процессом нулевому,
    countsrez[i]=C1;
    disprez[i]=i*C1; //смещение элементов в итоговом массиве
    disp[i]=i*C1*cols; //смещение элементов в исходном массиве
}
//Для size-C2
counts[size-C2]=(C1+1)*cols;
countsrez[size-C2]=C1+1;
disp[size-C2]=(size-C2)*C1*cols;
disprez[size-C2]=(size-C2)*C1;
//Для >size-C2
for (i=size-C2+1;i<size;i++) {
    counts[i]=(C1+1)*cols;
    countsrez[i]=C1+1;
    disp[i]=(size-C2)*C1*cols+(i-(size-C2))*(C1+1)*cols;
    disprez[i]=(size-C2)*C1+(i-(size-C2))*(C1+1);
}
//Распределяем строки матрицы по процессам
MPI_Scatterv(matrix, counts, disp, MPI_INT,
             matrix1, counts[rank], MPI_INT, 0, MPI_COMM_WORLD);
for(i=0;i<counts[rank]/cols;i++) rez[i]=0; //Обнуляем rez[]
//Вычисляем значение элемента результирующего вектора
for(i=0;i<counts[rank]/cols;i++)

```

Пример 6.9 (окончание). Умножение матрицы на вектор

```
for(j=0;j<cols;j++)
    rez[i]=matrix1[i*cols+j]*vector[j]+rez[i];
//Собираем результат в 0-процесс
MPI_Gatherv(rez, countsrez[rank], MPI_INT,result,
countsrez,
            disprez, MPI_INT, 0, MPI_COMM_WORLD);
if(!rank)
... //печать исходных данных и результатов 0-м процессом
... // Освобождение памяти
MPI_Finalize();
return 0;
}
```

Отметим также, что минимальный набор функций MPI и приемов работы с ними, достаточный для решения большинства задач с приемлемой эффективностью, не так велик. На практике часто оказывается, что владение эффективными приемами работы с небольшим количеством функций полезнее, чем поверхностное знание множества возможных вариантов реализации той или иной операции. А единственным способом, позволяющим обойтись малыми средствами, является тщательное проектирование и планирование всей структуры параллельной программы с учетом тех возможностей, которые в состоянии предоставить как кластер, так и MPI-библиотека, но *до*, а не *во* время написания программы.

Контрольные вопросы и задания

1. Какова специфика программирования для параллельных вычислительных систем с распределенной памятью?
2. Чем механизм передачи сообщений отличается от передачи данных через общую память?
3. Какие методы взаимодействия параллельных процессов используются при передаче сообщений?
4. Какие методы планирования процессов целесообразно применять для вычислительных систем с распределенной памятью?
5. Изложите общую концепцию построения MPI-программы.
6. Опишите синтаксис обрамляющих и информационных функций и структуру простейшей MPI-программы.
7. Опишите основные принципы декомпозиции на блоки регулярной многомерной сетки.
8. Изложите основные моменты распределения данных по узлам в MPI-приложении.

9. Опишите основные принципы организации вычислений в MPI. Введите понятия группы процессов, области связи коммуникатора.

10. Опишите основные MPI-функции создания групп и коммуникаторов, приведите примеры.

11. Дайте определение интра- и интеркоммуникаторов, приведите примеры их использования в MPI-программе.

12. Введите понятие виртуальной топологии в MPI, приведите примеры.

13. Введите понятие виртуальной топологии n -мерной решетки в MPI, укажите основные функции работы с такой топологией.

14. Введите понятие виртуальной топологии графа в MPI, укажите основные функции работы с такой топологией.

15. Перечислите базовые типы данных и основные предопределенные константы в MPI.

16. Как вводятся пользовательские типы данных в MPI? Приведите примеры использования пользовательских типов данных для эффективной реализации обменов в MPI-программе.

17. Опишите особенности трех признаков MPI-функций: блокирующие/неблокирующие, локальные, коллективные.

18. Опишите четыре режима отправки сообщения при парном обмене.

19. В чем особенность блокирующих отправок сообщений? Опишите синтаксис четырех основных функций. В чем особенность каждой из них? Ответ проиллюстрируйте примерами.

20. Какие MPI-функции информируют о характеристиках сообщения до его получения? Ответ проиллюстрируйте примерами.

21. В чем особенность неблокирующих отправок сообщений? Опишите синтаксис четырех основных функций. В чем особенность каждой из них? Ответ проиллюстрируйте примерами.

22. Как выполняется проверка выполнения обмена? Ответ проиллюстрируйте примерами.

23. Для чего используется функция совмещенных передачи и приема сообщений? Ответ проиллюстрируйте примерами.

24. Объясните суть алгоритма расчета фрагмента треугольника Паскаля. Как модифицировать алгоритм примера 5.6 таким образом, чтобы процесс насчитывал не один элемент, а квадратный блок элементов?

25. Опишите функционал приема данных в MPI. Ответ проиллюстрируйте примерами.

26. Как отправить и получить упакованное сообщение? Ответ проиллюстрируйте примерами.

27. Опишите функцию синхронизации барьером в MPI. Ответ проиллюстрируйте несколькими примерами ее использования.

28. Опишите глобальные функции связи в MPI. Ответ проиллюстрируйте несколькими примерами их использования.

29. Опишите глобальные функции приведения в MPI. Ответ проиллюстрируйте несколькими примерами их использования.

30. Как следует учитывать проблемы латентности и пропускной способности сети при проектировании MPI-программ?

Задачи

6.1. *Остров Сокровищ.* Шайка пиратов под предводительством Джона Сильвера высадилась на берег Острова Сокровищ. Несмотря на добытую карту старого Флинта, местоположение сокровищ по-прежнему остается загадкой, поэтому искать клад приходится практически на ощупь. Поскольку Сильвер ходит на деревянной ноге, то самому бродить по джунглям ему не с руки. Джон Сильвер поделил остров на участки, а пиратов – на небольшие группы. Каждой группе поручается искать клад на одном из участков, а сам Сильвер ждет на берегу. Пираты, обшарив свою часть острова, возвращаются к Сильверу и докладывают о результатах лично ему. Поскольку Сильвер крайне подозрителен, то группам пиратов под страхом смерти запрещено общаться между собой. Написать программу, моделирующую поведение пиратской шайки, используя метод передачи информации «точка – точка». Для общения каждой группы пиратов с Сильвером создать отдельный коммуникатор.

6.2. *Первая задача про китайский банк.* На маленькой улице Чжуань-Сю в городе Гонконг живут двести тысяч китайцев и находятся три банка. Каждый из этих банков принимает деньги от вкладчиков в трех валютах – китайских юанях, американских долларах и английских фунтах стерлингов. При этом если вкладчик хочет взять деньги в одном банке на улице Чжуань-Сю и положить в другой, то ему в первом банке выдается только расписка, которую он и относит во второй банк. В пятницу вечером банки подсчитывают, сколько денег и в какой валюте они должны соседям, и отправляют инкассаторов отнести эти деньги. Написать программу, моделирующую обмен деньгами в пятницу вечером на улице Чжуань-Сю, используя метод передачи информации «точка – точка».

6.3. *Вторая задача про китайский банк.* Решить задачу о китайских банках с дополнительным условием, что все пятничные расчеты банки проводят через Большой Банк, находящийся на улице Чжуань-Го.

6.4. *Криптономикон.* В секретном институте № 127, находящемся где-то в Сибири, разработана новейшая система шифровки данных для малого бизнеса. Система представляет собой ряд модулей, каждый из которых принимает на вход часть сообщения и шифрует его по определенной схеме. Какую именно часть текста шифрует каждый из модулей, задается

специальным ключом, который меняется каждый день. Написать программу, моделирующую работу шифровальной системы. В качестве шифров использовать прямые подстановки (вид шифра, когда каждому символу исходного текста ставится в соответствие какой-то символ шифрованного текста, причем всегда один и тот же). В качестве ключа использовать подстановку, нижняя строка которой задает номер модуля. Подаваемый для шифрования текст делится на равные части по количеству модулей, весь остаток получает тот модуль, который шифрует последнюю часть. Текст хранится в файле на диске и считывается управляющим процессом, затем передается согласно ключу соответствующему модулю для шифрования. Зашифрованный текст возвращается управляющему процессу, который выводит его в файл в необходимом порядке. Для соблюдения секретности создать по отдельному коммунитатору для передачи части текста от управляющего процесса каждому модулю. Использовать метод передачи информации «точка – точка».

6.5. *Криптономикон-2*. Дружественный секретный институт № 721, расположенный где-то на Урале, создал новейшую систему дешифрования новейшей системы шифрования, разработанной в секретном институте № 127, находящемся где-то в Сибири. Решить задачу, обратную задаче 6.4, написав программу-дешифровщик. Взяв в качестве входного сообщения шифрованный текст и ключ, программа должна восстанавливать исходный текст. Использовать парадигму «управляющий – рабочий», специальные коммунитаторы «управляющий процесс – модуль дешифровки», коллективные обмены.

6.6. *Первая задача про деньги*. Десять бухгалтеров сводят годовой баланс. Каждый из них отвечает за свою сторону жизни предприятия – зарплату, материальные расходы и т. п. – и точно знает, сколько доходов и расходов числится за ним. Бухгалтеры заполняют баланс, по очереди сообщая главному бухгалтеру данные своего раздела. Главный бухгалтер, сведя баланс, объявляет результат и везет отчет в налоговую инспекцию. Написать программу, моделирующую поведение бухгалтеров: а) используя метод передачи информации «точка – точка»; б) используя подходящую коллективную операцию.

6.7. *Аукцион*. Известный предприниматель и филантроп Джон Смит непосильным трудом на ниве экономических преступлений нажил огромное состояние и умер. Имущество досталось его сыну Джону Смиуту младшему, который, к сожалению, очень любил рулетку. Проиграв за полгода заводы, газеты и пароходы, приобретенные отцом, он для поправки дел решает распродать с аукциона вещи из отцовского особняка. Аукцион проводится следующим образом: каждый из участников втайне от остальных пишет свою цену на специальной карточке и отдает ее распорядителю.

Просмотрев карточки, распорядитель объявляет победителя. Написать программу, моделирующую проведение аукциона: а) используя метод передачи информации «точка – точка» и индивидуальные коммуникаторы для общения между каждым участником аукциона и распорядителем; б) используя подходящую коллективную операцию.

6.8. *Распределенное казино.* Два человека, находящихся в разных городах, играют в кости по скайпу. Чтобы избежать обмана, они сообщают результаты бросков своему товарищу, который определяет победителя и сообщает каждому из них о результате. Написать программу, моделирующую поведение игроков и посредника. Каждый процесс должен предоставить в виде файла протокол игры с указанием номеров партий, результатов бросков и победителя. Для общения каждого игрока с арбитром создать индивидуальные коммуникаторы.

6.9. *Произведение с дихотомией.* Дан вектор чисел $A[n]$, где n – произвольное число. Требуется найти значение выражения $C=A_1*A_2*\dots*A_n$, где A_i – i -й элемент вектора A . Число процессов P , используемых для решения задачи, должно быть не кратно числу n . Решить задачу: а) используя метод дихотомии и синхронный обмен сообщениями «точка – точка»; б) используя подходящую коллективную операцию. Свою часть вектора A каждый процесс читает из файла.

6.10. *Произведение с дихотомией с накоплением.* Решить задачу 6.9, вычисляя все промежуточные частичные произведения $C=A_1*A_2*\dots*A_i$. Число процессов P , используемых для решения задачи, должно быть не кратно числу n . Решить задачу: а) используя метод дихотомии и синхронный обмен сообщениями «точка – точка»; б) используя подходящую коллективную операцию. Свою часть вектора A каждый процесс читает из файла. Выходной файл должен содержать значение выражения C и все C_i .

6.11. *Кто быстрее?* Несколько процессов независимо друг от друга складывают по 1 000 000 чисел. Требуется определить номер самого быстрого процесса и время, затраченное каждым процессом на решение задачи. При решении задачи использовать барьерную синхронизацию.

6.12. Создать приложение с тремя процессами. Каждый процесс работает со своим массивом. Процессы сравнивают сумму элементов своего массива с суммами элементов других процессов и останавливаются, когда все три суммы равны между собой. Если суммы не равны, каждый процесс прибавляет единицу к одному элементу массива или отнимает единицу от одного элемента массива, затем снова проверяет условие равенства сумм. На момент остановки всех трех потоков суммы элементов массивов должны быть одинаковы.

6.13. Создать приложение с управляющим и тремя рабочими процессами. Каждый процесс работает с собственной очередью, каждый элемент

очереди содержит число и символ. Управляющий процесс составляет слово из тех символов, для которых равны между собой хотя бы два числа из трех соответствующих чисел в разных процессах.

6.14. Создать приложение с двумя процессами. Каждый процесс работает с массивом, содержащим нули и единицы. Процессы обмениваются элементами: первый процесс заменяет ноль единицей из массива второго потока, второй процесс заменяет единицу нулем из массива первого потока. Обмен происходит столько раз, сколько это возможно.

6.15. Создать приложение с четырьмя процессами. Каждый процесс работает с собственной строкой. Строки могут содержать только символы A, B, C, D. Процесс может поменять символ A на C, или C на A, или B на D, или D на B. Процессы останавливаются, когда общее количество символов A и B становится равным хотя бы для трех строк.

6.16. Реализовать барьер для n процессов по принципу дихотомии. Эффективность реализации сравнить с функцией `MPI_Barrier()`.

6.17. Реализовать симметричный барьер для n процессов. Эффективность реализации сравнить с функцией `MPI_Barrier()`.

6.18. Реализовать рассылку значения n процессам с помощью двухточечных обменов. Эффективность реализации сравнить с функцией `MPI_Bcast()`.

6.19. Реализовать сбор значений с n процессов с помощью двухточечных обменов (каждый процесс посылает одно значение). Эффективность реализации сравнить с функцией `MPI_Gather()`.

6.20. Реализовать рассылку массива значений на n процессов по принципу « i -е значение i -у процессу» с помощью двухточечных обменов. Эффективность реализации сравнить с функцией `MPI_Scatter()`.

6.21. Реализовать сбор данных с n процессов с помощью двухточечных обменов (каждый процесс посылает заранее заданное количество значений). Эффективность реализации сравнить с функцией `MPI_Gatherv()`.

6.22. Реализовать рассылку массива значений на n процессов с помощью двухточечных обменов по принципу «каждому процессу отправляется заранее заданное число значений». Эффективность реализации сравнить с функцией `MPI_Scatterv()`.

6.23. Реализовать функционал `MPI_Allgather()` с помощью двухточечных обменов. Эффективность реализации сравнить с функцией `MPI_Allgather()`.

6.24. Реализовать функционал `MPI_Alltoall()` с помощью двухточечных обменов. Эффективность реализации сравнить с функцией `MPI_Alltoall()`.

6.25. Реализовать сбор массива значений от n процессов на нулевой с помощью двухточечных обменов с выполнением какой-либо операции

приведения (сумма, минимум, среднее и пр.). Эффективность реализации сравнить с функцией `MPI_Reduce()`.

6.26. Реализовать сбор массива значений от n процессов на каждом процессе с помощью двухточечных обменов с выполнением какой-либо операции приведения (сумма, минимум, среднее и пр.) над каждым элементом массива. Эффективность реализации сравнить с функцией `MPI_Allreduce()`.

6.27. Реализовать распределение массива значений от n процессов с помощью двухточечных обменов с выполнением какой-либо операции приведения (сумма, минимум, среднее и пр.) таким образом, что бы на i -м процессе оказались приведенные i -е элементы массива. Эффективность реализации сравнить с функцией `MPI_Reduce_scatter()`.

6.28. Реализовать процедуру сканирования массива с приведением. Эффективность реализации сравнить с функцией `MPI_Scan()`.

6.29. Реализовать последовательный полный сдвиг по кольцу с помощью виртуальной топологии и функции `MPI_Cart_shift()`. Реализовать полный сдвиг по кольцу с помощью двухточечных обменов без использования виртуальной топологии. Сравнить эффективность двух реализаций.

6.30. Реализовать последовательный полный сдвиг по диагонали 2D-решетки с помощью виртуальной топологии и функции `MPI_Cart_shift()`. Реализовать полный сдвиг по диагонали 2D-решетки с помощью двухточечных обменов без использования виртуальной топологии. Сравнить эффективность двух реализаций.

6.31. Во входном файле `matrix.in` содержится размерность прямоугольной матрицы $m \times n$ (m – количество строк, n – количество столбцов) и сама матрица (хранится по строкам).

Необходимо распределить строки матрицы по P процессам (m не делится нацело на P) таким образом, чтобы количество строк у каждого процесса было почти одинаково (т. е. процессу отводится либо m/P строк, либо $m/P+1$).

Написать три программы:

- 1) блочное распределение;
- 2) циклическое слоистое распределение;
- 3) циклическое слоистое распределение с отражением.

Результат работы программы – файлы `proc0.out`, `proc1.out`, ..., `procP.out`, каждый из которых содержит количество строк, полученных процессом, сами строки с указанием их номера в глобальной нумерации.

Нельзя использовать пользовательские типы и упаковку данных. Задачу следует решить с помощью коллективных обменов.

6.32. Во входном файле `file.in` содержатся целые числа, количество которых не меньше удвоенного количества запущенных процессов P .

Первая часть программы. Нулевой процесс считывает все числа в массив. Затем рассылает всем процессам, включая себя, по несколько чисел с помощью функции `MPI_Scatterv()` следующим образом. Каждый процесс должен получить числа, начиная с числа, стоящего на месте, номер которого совпадает с рангом процесса, в количестве на единицу больше этого ранга (так, нулевой процесс получает нулевой член массива, первый – первый и второй члены, второй – второй, третий и четвертый и т. д.).

Получив числа, процессы выводят их в соответствующие выходные файлы: `proc0.out, proc1.out, ..., procP.out`.

Вторая часть программы. Все процессы, кроме нулевого, посылают полученные числа нулевому процессу неблокирующей функцией передачи сообщения `MPI_Isend()`, помечая сообщение тегом, равным своему рангу. Прежде чем завершиться, процесс дожидается удачного выполнения операции передачи.

Нулевой процесс получает сообщения в порядке их прихода. Для определения параметров сообщения перед приемом процесс узнает размер и тег пришедшего сообщения с помощью функции `MPI_Probe()`. После этого процесс получает сообщение с помощью функции блокирующего приема `MPI_Recv()`. Процесс добавляет запись в выходной файл `file.out` в следующем формате: «ранг процесса, приславшего сообщение (тег сообщения): строка полученных чисел».

6.33. Даны матрицы A размерностью $n1$ строк на $n2$ столбцов и матрица B размерностью $n2$ строк на $n3$ столбцов. Исходные данные хранятся в файлах на диске, подготовленные для каждого процессора (написать последовательную программу подготовки данных, входным параметром которой является число процессов).

Написать программу для P процессов ($2 \leq p \ll n1$, $p \ll n2$, $p \ll n3$, причем $p \% n1 \neq 0$, $p \% n2 \neq 0$, $p \% n3 \neq 0$), вычисляющую произведение матриц AB , собирающую результат в нулевом процессе, который выводит его в файл. Распределить вычислительную нагрузку на процессы максимально равномерно. Реализовать алгоритм:

- а) с топологией «кольцо» и парадигмой «взаимодействующие равные»;
 - б) с топологией «2D-решетка» и парадигмой «управляющий – рабочий»;
- 6.34–6.40. Задачи 4.7–4.23 решить следующим образом.

1. Написать последовательный алгоритм решения задачи. Замерить время, затрачиваемое на выполнение основного вычислительного процесса (исключая считывание данных, любые промежуточные выдачи, запись результатов).

2. Добавить директивы OpenMP для параллельного выполнения программы на одном узле SMP-узлов кластера. Провести исследование по оптимальному подбору параметров распараллеливания основных вычислительных циклов. Обосновать свой выбор, приведя графики ускорения и эффективности для различных наборов параметров распараллеливания.

3. Написать параллельную программу с помощью библиотеки MPI. Провести исследование ускорения и эффективности полученной программы. Сравнить результаты с аналогичными для OpenMP. Не теряя общности, для простоты реализации считать, что размерности задачи кратны количеству процессов.

4. Написать программу для SMP-узлов кластера, использующую технологию OpenMP для распараллеливания внутри SMP-узла и библиотеку MPI для распараллеливания между узлами. Провести исследование ускорения и эффективности, сравнить полученные результаты с аналогичными для программы из п. 3.

Тестирование задач проводить на небольшом объеме данных с заведомо проверяемым результатом. Исследование эффективности осуществлять на больших объемах данных, выбирая разумные размерности задачи, обеспечивающие, с одной стороны, хорошую масштабируемость (большой объем вычислений по сравнению с объемом пересылок или синхронизаций), с другой стороны, разумное время счета одного теста.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Антонов, А. С. Параллельное программирование с использованием технологии MPI: учеб. пособие / А. С. Антонов. – М.: Изд-во МГУ. – 2004. – 71 с.
2. Антонов, А. С. Параллельное программирование с использованием технологии OpenMP: учеб. пособие / А. С. Антонов. – М.: Изд-во МГУ. – 2009. – 77 с.
3. Бекон, Д. Операционные системы / Д. Бекон, Т. Харрис; пер. с англ. – СПб.: Питер; Киев: Издат. группа BHV, 2004. – 800 с.
4. Богачев, К. Ю. Основы параллельного программирования / К. Ю. Богачев. – М.: БИНОМ. Лаборатория знаний, 2010. – 342 с.
5. Боровский, А. Qt 4.7+ Практическое программирование на C++ / А. Боровский. – СПб.: БХВ-Петербург, 2012. – 496 с.
6. Вальковский, В. А. Распараллеливание алгоритмов и программ. Структурный подход / В. А. Вальковский. – М.: Радио и связь, 1989. – 176 с.
7. INMOST – программная платформа и графическая среда для разработки параллельных численных моделей на сетках общего вида: учеб. пособие / Ю. В. Василевский, И. Н. Коньшин, Г. В. Копытов, К. М. Терехов. – М.: Изд-во Моск. ун-та, 2013. – 144 с.
8. Вильямс, А. Системное программирование в Windows 2000 для профессионалов / А. Вильямс; пер. с англ. – СПб.: БХВ-Петербург, 2000. – 624 с.
9. Воеводин, В. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. – СПб.: БХВ-Петербург, 2002. – 608 с.
10. Воеводин, Вл. В. Вычислительное дело и кластерные системы / Вл. В. Воеводин, С. А. Жуматий. – М.: Изд-во МГУ, 2007. – 161 с.
11. Высокопроизводительные вычисления на кластерах: учеб. пособие / под ред. А. В. Старченко. – Томск: Изд-во Том. ун-та, 2008. – 198 с.
12. Гергель, В. П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем: учебн. пособие / В. П. Гергель. – Изд-во Московского университета, 2010. – 544 с.
13. Гергель, В. П. Основы параллельных вычислений для многопроцессорных вычислительных систем: учеб. пособие / В. П. Гергель, Р. Г. Стронгин. – Н. Новгород: Изд-во Нижегород. ун-та, 2003. – 184 с.
14. Гергель, В. П. Теория и практика параллельных вычислений / В. П. Гергель. – М.: Интернет-университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007. – 424 с.

15. Даутов, Р. З. Введение в метод конечных элементов: учеб. пособие / Р. З. Даутов, М. М. Карчевский. – Казань: Изд-во Казан. ун-та им. В. И. Ульянова-Ленина, 2004. – 239 с.
16. Дейкстра, Э. Взаимодействующие последовательные процессы / Э. Дейкстра; пер. с англ. // Языки программирования; под ред. Ф. Женюи. – М.: Мир, 1972. – 406 с.
17. Дементьева, Е. В. Эффективность численного моделирования на кластерных системах распространения поверхностных волн / Е. В. Дементьева, Е. Д. Каропова, А. В. Малышев // Вестн. НГУ. – 2011. – Т. 9, № 1. – С. 11–20. – (Информационные технологии).
18. Ильин, В. П. О стратегиях распараллеливания в математическом моделировании / В. П. Ильин // Программирование. – 1999. – № 1. – С. 41–46.
19. Ильин, В. П. Методы и технологии конечных элементов / В. П. Ильин. – Новосибирск: Изд-во ИВМиМГ СО РАН, 2007. – 371 с.
20. Ильин, В. П. Параллельные алгоритмы для больших прикладных задач: проблемы и технологии / В. П. Ильин // Автометрия. – 2007. – Т. 43, № 2. – С. 3–21.
21. Ильин, В. П. Проблемы высокопроизводительных технологий решения больших разреженных СЛАУ / В. П. Ильин // Вычислительные методы и программирование. – 2009. – Т. 10, № 1. – С. 130–136.
22. Каропова, Е. Д. Параллельная реализация МКЭ для начально-краевой задачи мелкой воды / Е. Д. Каропова, В. В. Шайдуров // Вычислительные технологии. – 2009. – Т. 14, № 6. – С. 45–57.
23. Карпов, В. Е. Введение в операционные системы: курс лекций / В. Е. Карпов, К. А. Коньков. – 2-е изд. – М.: ИНТУИТ.РУ, 2005. – 536 с.
24. Касперски, К. Техника оптимизации программ. Эффективное использование памяти / К. Касперски. – СПб.: БХВ-Петербург, 2003. – 464 с.
25. Корнеев, В. Д. Параллельные вычислительные системы / В. Д. Корнеев. – М.: Нолидж, 1999. – 320 с.
26. Корнеев, В. Д. Параллельное программирование в MPI / В. Д. Корнеев. – М.; Ижевск: Ин-т компьютерных исследований, 2003. – 304 с.
27. Корнеев, В. Д. Параллельное программирование в MPI / В. Д. Корнеев. – Новосибирск: Изд-во СО РАН, 2000. – 213 с.
28. Инструменты параллельного программирования в системах с общей памятью: учебник / К. В. Корняков, В. Д. Кустикова, И. Б. Меев и др. – М.: Изд-во Моск. ун-та, 2010. – 272 с.
29. Крюков, В. А. Разработка параллельных программ для вычислительных кластеров и сетей / В. А. Крюков // Информационные технологии и вычислительные системы. – 2003. – № 1–2. – С. 42–61.

30. Малышкин, В. Э. Введение в параллельное программирование мультимикомпьютеров / В. Э. Малышкин. – М.; Новосибирск, 2003. – 268 с.
31. Марчук, Г. И. Введение в проекционно-сеточные методы / Г. И. Марчук, В. И. Агошков. – М.: Наука, 1981. – 416 с.
32. Немнюгин, С. А. Параллельное программирование для много-процессорных вычислительных систем / С. А. Немнюгин, О. Л. Стесик. – СПб.: БХВ-Петербург, 2002. – 400 с.
33. Ортега, Дж. Введение в параллельные и векторные методы решения линейных систем / Дж. Ортега; пер. с англ. – М.: Мир, 1991. – 367 с.
34. Рихтер, Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер; пер. с англ. – СПб.: Русская Редакция, 2000. – 752 с.
35. Старченко, А. В. Методы параллельных вычислений: учебник / А. В. Старченко, В. Н. Берцун. – Томск: Изд-во Том. ун-та, 2013. – 223 с.
36. Стивенс, У. Unix: Разработка сетевых приложений / У. Стивенс; пер. с англ. – СПб.: Питер, 2004. – 1086 с. – (Мастер-класс).
37. Стивенс, У. UNIX: Взаимодействие процессов / У. Стивенс; пер. с англ. – СПб.: Издат. дом «Питер», 2002. – 576 с.
38. Сьярле, Филипп. Метод конечных элементов для эллиптических задач / Ф. Сьярле; пер. с англ. – М.: Мир, 1980. – 512 с.
39. Таненбаум, Э. Архитектура компьютера / Э. Таненбаум; пер. с англ. – СПб.: Питер, 2013. – 816 с. – (Классика Computer Science).
40. Таненбаум, Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, Ван Стеен; пер. с англ. – СПб.: Питер, 2003. – 877 с. – (Классика Computer Science).
41. Таненбаум, Э. Современные операционные системы. Modern Operating Systems / Э. Таненбаум, Х. Бос; пер. с англ. – СПб.: Питер, 2016. – 1120 с. – (Классика Computer Science).
42. Таненбаум, Э. Операционные системы / Э. Таненбаум, А. Вудхал; пер. с англ. – СПб.: Питер, 2006. – 576 с.
43. Хокни, Р. Параллельные ЭВМ. Архитектура, программирование и алгоритмы / Р. Хокни, К. Джессхоуп; пер. с англ. – М.: Радио и связь, 1986. – 392 с.
44. Хьюз, Камерон. Параллельное и распределенное программирование на C++ / Хьюз Камерон, Хьюз Трейси; пер. с англ. – М.: Издат. дом «Вильямс», 2004. – 672 с.
45. Чан, Теренс. Системное программирование на C++ для UNIX / Теренс Чан; пер. с англ. – СПб.: БХВ-Петербург, 1999. – 592 с.
46. Шлее, М. Qt 5.3. Профессиональное программирование на C++ / М. Шлее. – СПб.: БХВ-Петербург, 2015. – 928 с.

47. Шоу, А. Логическое проектирование операционных систем / А. Шоу; пер. с англ. – М.: Мир, 1981. – 360 с.
48. Эндрюс, Грегори Р. Основы многопоточного, параллельного и распределенного программирования / Грегори Р. Эндрюс; пер. с англ. – М.: Издат. дом «Вильямс», 2003. – 512 с.
49. Юлдашев, А. В. Исследование влияния конкуренции за каналы передачи данных коммуникационной среды кластерной системы на время выполнения MPI-программ / А. В. Юлдашев // Научный сервис в сети Интернет: суперкомпьютерные центры и задачи: Труды Междунар. суперкомпьютерной конф. (20–25 сент. 2010 г., г. Новороссийск). – М.: Изд-во МГУ, 2010. – С. 560–567.
50. Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities / G. Amdahl // AFIPS Conference Proceedings. – Washington, D. C.: Thompson Books. – 1967. – V. 30. – P. 483–485.
51. Andrews G. R. Foundations of Multithreaded, Parallel, and Distributed Programming / G. R. Andrews. – Reading.: Addison-Wesley, 2000. – 664 p.
52. Bertsekas, D. P. Parallel and distributed Computation: Numerical Methods / D. P. Bertsekas, J. N. Tsitsiklis. – USA, Nashua: Athena Scientific, 1997. – 718 p.
53. Butenhof, D. R. Programming with POSIX Threads / D. R. Butenhof. – Boston, MA: Addison-Wesley Professional., 1997. – 383 p.
54. Parallel Programming in OpenMP / R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. – San-Diego, USA: Academic Press, 2000. – 231 p.
55. Cohen, Aaron. Win32 Multithreaded Programming / Aaron Cohen, Mike Woodring. – O'Reilly Media, 1998. – 705 p.
56. Culler, David E. Parallel Computer Architecture: A Hardware / Software Approach / D. E. Culler, J. P. Singh, A. Gupta. – San Francisco: Morgan Kaufmann Publisher, 1998. – 1025 p.
57. LogP Performance Assessment of Fast Network Interfaces / D. E. Culler, L. T. Liu, R. P. Martin, Ch. O. Yoshikaw // IEEE Micro. – 1996. – V. 16(1). – P. 35–43.
58. Gove, Darryl. Multicore Application Programming For Windows, Linux, and Oracle Solaris / Darryl Gove. – Addison-Wesley Professional, 2011. – 441 p.
59. Numerical Linear Algebra for High Performance Computers / J. J. Dongarra, L. S. Duff, D. C. Sorensen, H. A. V. Vorst. – Soc. For Industrial & Applied Math., 1999. – 336 p. (Software, Environments, Tools).
60. Flynn, M. J. Very high speed computers / M. J. Flynn // Proc. IEEE. – 1966. – V. 54 (12). – P. 1901–1909.

-
61. Flynn, M. J. Some Computer Organizations and Their Effectiveness / M. J. Flynn // *Computers, IEEE Transactions on*. – 1972. – V. C-21 (9). – P. 948–960.
62. Solving Problems on Concurrent Processors / G. C. Fox, et al. // New Jersey: Prentice Hall Inc., Englewood Cliffs. – 1988. – V. 1. – P. 187–200.
63. Group, W. Using MPI. Portable Parallel Programming with the Message-Passing Interface / W. Group, E. Lusk, A. Skjellum – Cambridge: MIT Press, 1994. – 275 p.
64. High Performance Cluster Computing. – Vol. 1: Architectures and Systems. – Vol. 2: Programming and Applications / Red. R. Buyya. – New Jersey: Prentice Hall PTR, Prentice-Hall Inc., 1999.
65. Hockney, R. W. The communication challenge for MPP: Intel Paragon and Meiko CS2 / R. W. Hockney // *Parallel Computing*. – 1998. – V. 20. – P. 389–398.
66. Jimack, P. K. Developing Parallel Finite Element Software Using MPI / P. K. Jimack, N. Touheed // *High Performance Computing for Computational Mechanics*. – 2000. – P. 15–38.
67. Karepova, E. The numerical solution of data assimilation problem for shallow water equations / E. Karepova, V. Shaidurov, E. Dementyeva // *Int. J. of Num. Analysis and Modeling, Series B*. – 2011. – V. 2 (2–3). – P. 167–182.
68. Lamport, L. A new approach to proving the correctness of multiprocess programs / L. Lamport. – *ACM Trans. on Prog. Languages and Systems*. – 1979. – V. 1 (1). – P. 84–97.
69. Mahinthakumar, G. A Hybrid MPI-OpenMP Implementation of an Implicit Finite-Element Code on Parallel Architectures / G. Mahinthakumar, F. Saied // *International Journal of High Performance Computing Applications*. – 2002. – V. 16(4). – P. 371–393.
70. Mattson, Timothy G. Patterns for Parallel Programming / Timothy G. Mattson, Berna L. Massingill, Beverly A. Sanders. – Addison-Wesley Professional, 2004. – 355 p.
71. Pantale, O. Parallelization of an object-oriented FEM dynamics code: influence of the strategies on the Speedup / O. Pantale // *Advances in Engineering Software*. – 2005. – V. 36(6). – P. 361–373.
72. Patterson, D. A. Computer Architecture: A Quantitative Approach. 5th Edition / D. A. Patterson, J. L. Hennessy. – San Francisco: Morgan Kaufmann Publisher, 2011. – 856 p.
73. Reinders, James. Intel Threading Building Blocks : Outfitting C++ for Multi-Core Processor Parallelism / J. Reinders. – O'Reilly Media, 2007. – 336 p.
74. Smith, I. M. Programming the finite element method. 5th Edition / I. M. Smith, D. V. Griffiths, L. Margetts. – Chichester: Wiley, 2004. – 682 p.

75. Vargas-Felix, M. Solution of Finite Element Problems Using Hybrid Parallelization with MPI and OpenMP / M. Vargas-Felix, S. Botello-Rionda // Acta Universitaria. – 2012. – V. 22(7). – P. 14–24.

76. Vollaie, C. Parallel computing for the finite element method / C. Vollaie, L. Nicolas, A. Nicolas // The European Physical Journal Applied Physics. – 1998. – V. 1(3). – P. 305–314.

Информационные ресурсы в сети Интернет

77. Информационно-аналитические материалы по параллельным вычислениям: <http://www.parallel.ru>.

78. Информационно-аналитические материалы Центра компьютерного моделирования Нижегородского госуниверситета им. Н. И. Лобачевского: <http://www.software.unn.ac.ru/ccam>.

79. Информационные материалы по MPI: <http://www.mpi-forum.org>

80. Информационные материалы по OpenMP: <http://openmp.org>.

81. Материалы по многопоточному программированию и стандарту POSIX. Multi-Threaded Programming With POSIX Threads: <http://users.act-com.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html>.

82. Национальный открытый университет ИНТУИТ. Курсы по суперкомпьютерным технологиям: http://www.intuit.ru/studies/courses?service=0&option_id=95&service_path=1.

83. Портал «Top 500® Supercomputers Sites»: <http://www.top500.org>

84. Сайт компании Intel: <http://www.intel.com>.

85. Сайт, посвященный библиотеке LinPack: <http://www.netlib.org/linpack>.

86. Сайт профессора А. И. Легалова с публикациями по моделям параллельных вычислений и методам параллельного программирования: <http://www.softcraft.ru>.

87. Спецификации стандарта MPI: <http://www.mpi-forum.org/docs/docs.html>.

88. Спецификации стандарта OpenMP: <http://openmp.org/wp/openmp-specifications/>.

89. Многопроцессорные вычислительные системы и параллельное программирование: учеб. курс / под ред. В. П. Гергеля // Нижегород. гос. ун-т им. Н. И. Лобачевского, Факультет вычислительной математики и кибернетики: http://www.software.unn.ac.ru/ccam/mskurs/RUS/HTML/cs338_ppr_materials.htm.

90. Patrick Lam. Course ECE 459: Programming for Performance: <http://patricklam.ca/p4p/notes/>.

Учебное издание

Карпова Евгения Дмитриевна

**ОСНОВЫ МНОГОПОТОЧНОГО
И ПАРАЛЛЕЛЬНОГО
ПРОГРАММИРОВАНИЯ**

Учебное пособие

Редактор *Я. Н. Лысь*

Корректор *В. Р. Наумова*

Компьютерная верстка *Н. Г. Дербенёвой*

Подписано в печать 23.12.2016. Печать плоская. Формат 60×84/16
Бумага офсетная. Усл. печ. л. 22,25. Тираж 500 экз. Заказ № 4261

Библиотечно-издательский комплекс
Сибирского федерального университета
660041, Красноярск, пр. Свободный, 82а
Тел. (391) 206-26-67; <http://bik.sfu-kras.ru>
E-mail: publishing_house@sfu-kras.ru

**В Библиотечно-издательском комплексе СФУ
вам быстро и качественно выполнят следующие виды
издательских работ:**

- редактирование**
- корректура**
- художественное оформление**
- компьютерная верстка**

Наш адрес:
660041, г. Красноярск, пр. Свободный, 82а, к. 0108
Тел. (391) 206-26-67 – отдел приема и сопровождения заказа

Для заметок