

Министерство образования и науки Российской Федерации

Нижегородский государственный университет

им. Н.И. Лобачевского

**В.П. Гергель**

**ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ  
МНОГОЯДЕРНЫХ МНОГОПРОЦЕССОРНЫХ СИСТЕМ**

Издательство Нижегородского госуниверситета

Нижний Новгород

2010

**УДК 004.42.032.24(07)**

**ББК 32.973.2-018.2.7**

**Г 37**

Г 37 Гергель В.П. *Высокопроизводительные вычисления для многоядерных многопроцессорных систем.* Учебное пособие – Нижний Новгород; Изд-во ННГУ им. Н.И.Лобачевского, 2010

В работе излагается учебный материал, достаточный для успешного начала работ в области параллельного программирования. Для этого в пособии дается краткая характеристика принципов построения параллельных вычислительных систем, рассматриваются математические модели параллельных алгоритмов и программ для анализа эффективности параллельных вычислений, приводятся примеры конкретных параллельных методов для решения типовых задач вычислительной математики.

Учебное пособие предназначено для широкого круга студентов, аспирантов и специалистов, желающих изучить и практически использовать параллельные компьютерные системы для решения вычислительно трудоемких задач.

Рекомендовано Советом учебно-методическим объединением классических университетов по прикладной математике и информатике.

Подготовка учебника была выполнена в рамках работ программы развития Нижегородского университета как Национального исследовательского университета.

ISBN 5-85746-602-4

Гергель В.П., 2010

# **УЧЕБНИК**

## **"ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ МНОГОЯДЕРНЫХ МНОГОПРОЦЕССОРНЫХ СИСТЕМ"**

### **Аннотация**

Потребность решения сложных прикладных задач с большим объемом вычислений и принципиальная ограниченность максимального быстродействия "классических" - по схеме фон Неймана - ЭВМ привели к появлению многопроцессорных вычислительных систем (МВС). Особую значимость параллельные вычисления приобрели с переходом компьютерной индустрии на массовый выпуск многоядерных процессоров.

Суперкомпьютерный технологии и высокопроизводительные вычисления с использованием параллельных вычислительных систем становятся важным фактором научно-технического прогресса; их применение принимает всеобщий характер.

Знание современных тенденций развития ЭВМ и аппаратных средств для достижения параллелизма, умение разрабатывать модели, методы и программы параллельного решения задач обработки данных следует отнести к числу важных квалификационных характеристик современного специалиста по прикладной математике, информатике и вычислительной технике

Цель учебника состоит в изложении основных понятий параллельных вычислений, необходимых для последующего изучения моделей, методов и технологий параллельного программирования.

В рамках учебника проводится изучение следующего набора тем:

- Краткий обзор параллельных вычислительных систем и их классификация,
- Общая характеристика многопроцессорных вычислительных систем,
- Оценка эффективности параллельных вычислений,
- Анализ сложности вычислений и оценка возможности распараллеливания,
- Изложение технологии OpenMP,
- Общая схема разработки параллельных методов,
- Методы параллельных вычислений для задач вычислительной математики (матричные вычисления, решение систем линейных уравнений, сортировка, обработка графов, уравнения в частных производных, многоэкстремальная оптимизация)
- Программная система Паралаб для изучения и исследования методов параллельных вычислений.

При изложении учебного материала в учебнике предполагается наличие общих (начальных) знаний в области программирования.

## **Введение**

Среди многих закономерностей компьютерного мира можно выделить, пожалуй, самый непреложный закон – неуклонное и непрерывное повышение производительности компьютерных систем. Необходимость повышения быстродействия вычислений во многом обуславливается практической потребностью – на более высокопроизводительных компьютерах можно решать сложные вычислительно-трудоемкие научно-технические задачи более быстро. Кроме того, возрастание производительности компьютерной техники позволяет повышать сложность решаемых задач и постоянно расширять круг исследуемых проблем. Сложность и трудоемкость задач, которые в настоящее время решаются с помощью компьютеров, являются просто огромными и на порядки превышают наши представления 5-10 давности.

Росту производительности компьютеров способствует и постоянное совершенствование технологий создания средств компьютерной техники. В сфере производства компьютеров обязательным требованием является соблюдение закона Мура, в соответствии с которым «производительность вычислительных систем должна удваиваться каждые 18 месяцев». Совсем до недавнего времени подобный рост производительности обеспечивался во многом повышением тактовой частоты основных вычислительных элементов компьютеров – процессоров. Но возможности такого подхода оказались не безграничными – после некоторого рубежа дальнейшее увеличение тактовой частоты требует значительных технологических усилий, сопровождается существенным ростом энергопотребления и наталкивается на непреодолимые проблемы теплорегуляции. В таких условиях практически неизбежным явилось кардинальное изменение основного принципа производства компьютерной техники – вместо создания сложных высокочастотных «монолитных» процессоров новая «генеральная» линия стала состоять в разработке «составных» процессоров, состоящих из множества равноправных и сравнительно простых вычислительных элементов – ядер. Максимальная производительность процессоров в этом случае является равной сумме производительности вычислительных ядер, входящих в процессоры. Тем самым, «упаковывая» в рамках процессоров все большее количество ядер, можно добиваться роста производительности без «проблематичного» повышения тактовых частот.

Итак, вместо «гонки частот» наступила эпоха многоядерных процессоров. Уже в настоящее время массово выпускаемые процессоры содержат от 4 до 8 ядер; компании-производители компьютерной техники заявляют о своих планах по выпуску 12-16-ядерных процессоров. В ближайшей перспективе процессоры с сотнями и тысячами ядер.

При этом важно понимать, что переход к многоядерности одновременно знаменует и наступление эры параллельных вычислений. На самом деле, задействовать вычислительный потенциал многоядерных процессоров можно только, если осуществить разделение выполняемых вычислений на информационно независимые части и организовать выполнение каждой части вычислений на разных ядрах. Подобный подход позволяет выполнять необходимые вычисления с меньшими затратами времени, и возможность получения максимального ускорения ограничивается только числом имеющихся ядер и количеством "независимых" частей в выполняемых вычислениях. Параллельные вычисления становятся неизбежными и повсеместными.

Однако необходимость организации параллельных вычислений приводит к повышению сложности эффективного применения многоядерных компьютерных систем. Для проведения параллельных вычислений необходимо "параллельное" обобщение традиционной - последовательной - технологии решения задач на ЭВМ. Так, численные методы в случае многоядерности должны проектироваться как системы параллельных и взаимодействующих между собой процессов, допускающих исполнение на независимых вычислительных ядрах. Применяемые алгоритмические языки и системное программное обеспечение должны обеспечивать создание параллельных программ, организовывать синхронизацию и взаимоисключение асинхронных процессов и т.п.

Все перечисленные проблемы организации параллельных вычислений увеличивают существующий разрыв между вычислительным потенциалом современных компьютерных систем и имеющимся алгоритмическим и программным обеспечением применения компьютеров для решения сложных задач. И, как результат, устранение или, по крайней мере, сокращение этого разрыва является одной из наиболее значимых задач современной науки и техники.

В данной книге изложен учебных материал, достаточный для успешного начала работ в области параллельного программирования для современных высокопроизводительных многоядерных многопроцессорных систем с общей разделяемой памятью:

- В главе 1 приводится обзор архитектуры современных многоядерных процессоров;
- В главе 2 проводится анализ эффективности параллельных вычислений для оценивания получаемого ускорения вычислений и степени использования всех возможностей компьютерного оборудования при параллельных способах решения задач;
- В главе 3 рассматриваются общие принципы разработки параллельных алгоритмов для решения сложных вычислительно-трудоемких задач;
- В главе 4 излагаются основы параллельного программирования (понятие процессов и потоков, организация взаимодействия и взаимоисключения потоков, методы синхронизации, классические задачи параллельного программирования и др.);
- В главе 5 рассматриваются вопросы параллельного программирования с использованием технологии OpenMP для многоядерных многопроцессорных систем с общей разделяемой памятью;
- В главах 6-12 рассматриваются вопросы создания и развития параллельных алгоритмов для решения прикладных задач в разных областях практических приложений; в этих же лекциях представлены возможные программные реализации рассматриваемых алгоритмов;
- В заключительной главе 13 рассматривается учебно-исследовательская программная система Параллельная Лаборатория (ПараЛаб), которая может быть использована для организации лабораторного практикума для изучения и исследования эффективности параллельных алгоритмов.

Подготовка настоящего учебника выполнена на основе лекционного материала учебного курса "Многопроцессорные вычислительные системы и параллельное программирование", читаемого с 1996 г. в Нижегородском государственном университете на факультете вычислительной математики кибернетики для студентов, обучаемых по специальности "Прикладная математика и информатика" и новому

направлению подготовки "Информационные системы"  
(<http://www.software.unn.ac.ru/ccam/?doc=98>).

Значительная часть работ по развитию учебно-методического и программного обеспечения данного курса была выполнена в 2007-2009 гг. в учебно-исследовательской лаборатории Информационные технологии (ИТЛаб), созданной в Нижегородском университете при поддержке компании Интел. За прошедший период учебные материалы активно использовалось в разных формах подготовки и переподготовки кадров в области современных компьютерных технологий. Материалы учебника применялись в проводимых в Нижегородском университете при поддержке компании Интел в 2007-2009 гг. Зимних школах по параллельному программированию и вошли в учебную программу Всероссийской молодежной школы «Суперкомпьютерные технологии в образовании, науке и промышленности» (Нижний Новгород, октябрь 2009).

Данное издание является переработанным вариантом учебника «Теория и практика параллельных вычислений», изданного в рамках Интернет-университета информационных технологий [10] и изначально ориентированного на многопроцессорные компьютерные системы с распределенной памятью.

Автор данного курса лекций считает своим приятным долгом выразить благодарность за совместную успешную работу и дружескую поддержку своим коллегам – преподавателям кафедры математического обеспечения ЭВМ факультета вычислительной математики и кибернетики Нижегородского госуниверситета. Большое практическое содействие в подготовке материалов учебников оказали Лабутина А., Козинов Е., Сысоев А.

Подготовка учебника была поддержана компанией Интел в числе победителей конкурса учебно-образовательных материалов программы Intel Multicore Curriculum Program по созданию учебно-методического обеспечения для многоядерных вычислительных систем.

# **ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ МНОГОЯДЕРНЫХ МНОГОПРОЦЕССОРНЫХ СИСТЕМ**

**В.П. Гергель**

## **Содержание**

Глава 1. Обзор архитектуры современных многоядерных процессоров	1
1.1. Параллелизм как основа высокопроизводительных вычислений	3
1.1.1. Симметрическая мультипроцессорность.	4
1.1.2. Одновременная многопотоковость.	6
1.1.3. Многоядерность.	7
1.2. Многоядерность – два, четыре, восемь – кто больше?	9
1.2.1. Процессоры Intel® Core™ и Intel® Xeon®.	9
1.2.2. Процессоры AMD Phenom™ и AMD Opteron™.	11
1.2.3. Процессоры IBM Power6.	12
1.2.4. Процессоры PowerXCell™ 8i.	13
1.2.5. Процессоры Sun UltraSPARC T1 и Sun UltraSPARC T2.	14
1.3. Ускорители вычислений	16
1.3.1. Ускоритель ClearSpeed™ Advance™ X620.	16
1.3.2. Настольный суперкомпьютер NVIDIA® Tesla™ D870.	17
1.4. Персональные мини-кластеры	18
1.4.1. Персональный суперкомпьютер T-Forge Mini.	19
1.4.2. Мини-кластер T-Edge Mini.	20
1.5. Краткий обзор раздела	20
1.6. Обзор литературы	21
1.7. Контрольные вопросы	21
1.8. Задачи и упражнения	22
Глава 2. Моделирование и анализ параллельных вычислений	2
2.1. Модель вычислений в виде графа "операции-операнды"	2
2.2. Описание схемы параллельного выполнения алгоритма	3
2.3. Определение времени выполнения параллельного алгоритма	4
2.4. Показатели эффективности параллельного алгоритма	6
2.5. Вычисление частных сумм последовательности числовых значений	7
2.5.1. Последовательный алгоритм суммирования	8
2.5.2. Каскадная схема суммирования	8
2.5.3. Модифицированная каскадная схема	10
2.5.4. Вычисление всех частных сумм	11
2.6. Оценка максимально достижимого параллелизма	12
2.7. Анализ масштабируемости параллельных вычислений	14
2.8. Краткий обзор раздела	15
2.9. Обзор литературы	16
2.10. Контрольные вопросы	17

2.11. Задачи и упражнения	18
Глава 3. Принципы разработки параллельных методов	2
3.1. Моделирование параллельных программ	4
3.2. Этапы разработки параллельных алгоритмов	5
3.2.1. Разделение вычислений на независимые части	6
3.2.2. Выделение информационных зависимостей	8
3.2.3. Масштабирование набора подзадач	9
3.2.4. Распределение подзадач между вычислительными элементами	10
3.3. Параллельное решение гравитационной задачи N тел	11
3.3.1. Разделение вычислений на независимые части	12
3.3.2. Выделение информационных зависимостей	12
3.3.3. Масштабирование и распределение подзадач по процессорам	12
3.4. Краткий обзор раздела	12
3.5. Обзор литературы	13
3.6. Контрольные вопросы	13
3.7. Задачи и упражнения	13
Глава 4. Основы параллельного программирования	2
4.1. Основные понятия	2
4.1.1. Концепция процесса	2
4.1.2. Определение потока	3
4.1.3. Понятие ресурса	3
4.1.4. Организация параллельных программ как системы потоков	4
4.2. Взаимодействие и взаимоисключение потоков	7
4.2.1. Разработка алгоритма взаимоисключения	7
Вариант 1 – Жесткая синхронизация	8
Вариант 2 – Потеря взаимоисключения	8
Вариант 3 – Возможность взаимоблокировки	9
Вариант 4 – Бесконечное откладывание	10
Алгоритм Деккера	10
4.2.2. Семафоры	11
4.2.3. Мониторы	12
4.3. Синхронизация потоков	15
4.3.1. Условные переменные	15
4.3.2. Барьерная синхронизация	15
4.4. Взаимоблокировка потоков	16
4.4.1. Модель программы в виде графа "поток-ресурс"	17
4.4.2. Описание возможных изменений состояния программы	18
4.4.3. Обнаружение и исключение тупиков	19
4.5. Классические задачи синхронизации	19
4.5.1. Задача "Производители-Потребители"	20
4.5.2. Задача "Читатели-Писатели"	21
4.5.3. Задача "Обедающие философы"	22
4.5.4. Задача "Спящий парикмахер"	24

4.6. Методы повышения эффективности параллельных программ	25
4.6.1. Оптимизация количества потоков	26
4.6.2. Минимизация взаимодействия потоков	26
4.6.3. Оптимизация работы с памятью	27
Обеспечение однозначности кэш-памяти	28
Уменьшение миграции потоков между ядрами/процессорами	28
Устранение эффекта ложного разделения данных	28
4.6.4. Использование потоко-ориентированных библиотек	28
4.7. Краткий обзор раздела	29
4.8. Обзор литературы	29
4.9. Контрольные вопросы	30
4.10. Задачи и упражнения	31
Глава 5. Параллельное программирование с использованием OpenMP	2
5.1. Основы технологии OpenMP	4
5.1.1. Понятие параллельной программы	4
5.1.2. Организация взаимодействия параллельных потоков	5
5.1.3. Структура OpenMP	6
5.1.4. Формат директив OpenMP	6
5.2. Выделение параллельно-выполняемых фрагментов программного кода	7
5.2.1. Директива parallel для определения параллельных фрагментов	7
5.2.2. Пример первой параллельной программы	7
5.2.3. Основные понятия параллельной программы: фрагмент, область, секция	8
5.2.4. Параметры директивы parallel	9
5.2.5. Определение времени выполнения параллельной программы	9
5.3. Распределение вычислительной нагрузки между потоками (распараллеливание по данным для циклов)	10
5.3.1. Управление распределением итераций цикла между потоками	12
5.3.2. Управление порядком выполнения вычислений	13
5.3.3. Синхронизация вычислений по окончании выполнения цикла	13
5.3.4. Введение условий при определении параллельных фрагментов (параметр if директивы parallel)	14
5.4. Управление данными для параллельно-выполняемых потоков	14
5.4.1. Определение общих и локальных переменных	15
5.4.2. Совместная обработка локальных переменных (операция редукции)	16
5.5. Организация взаимоисключения при использовании общих переменных	17
5.5.1. Обеспечение атомарности (неделимости) операций	17
5.5.2. Использование критических секций	18
5.5.3. Применение переменных семафорного типа (замков)	19
5.6. Распределение вычислительной нагрузки между потоками (распараллеливание по задачам при помощи директивы sections)	20
5.7. Расширенные возможности OpenMP	22
5.7.1. Определение однопотоковых участков для параллельных фрагментов (директивы single и master)	22
5.7.2. Выполнение барьерной синхронизации (директива barrier)	23

5.7.3. Синхронизация состояния памяти (директива flush)	23
5.7.4. Определение постоянных локальных переменных потоков (директива threadprivate и параметр copyin директивы parallel)	24
5.7.5. Управление количеством потоков	25
5.7.6. Задание динамического режима при создании потоков	26
5.7.7. Управление вложенностью параллельных фрагментов	26
5.8. Дополнительные сведения	26
5.8.1. Разработка параллельных программ с использованием OpenMP на алгоритмическом языке Fortran	27
5.8.2. Сохранение возможности последовательного выполнения программы	28
5.8.3. Краткий перечень компиляторов с поддержкой OpenMP	29
5.9. Краткий обзор раздела	29
5.10. Обзор литературы	30
5.11. Контрольные вопросы	31
Задачи и упражнения	32
Приложение: Справочные сведения об OpenMP	33
П1. Сводный перечень директив OpenMP	33
П2. Сводный перечень параметров директив OpenMP	34
П3. Сводный перечень функций OpenMP	36
П4. Сводный перечень переменных окружения OpenMP	37
Глава 6. Параллельные методы умножения матрицы на вектор	1
6.1. Введение	1
6.2. Принципы распараллеливания	2
6.3. Постановка задачи	3
6.4. Последовательный алгоритм	4
6.5. Умножение матрицы на вектор при разделении данных по строкам	5
6.5.1. Выделение информационных зависимостей	5
6.5.2. Масштабирование и распределение подзадач по вычислительным элементам	6
6.5.3. Программная реализация	6
6.5.4. Анализ эффективности	7
6.5.5. Результаты вычислительных экспериментов	11
6.6. Умножение матрицы на вектор при разделении данных по столбцам	14
6.6.1. Определение подзадач и выделение информационных зависимостей	14
6.6.2. Масштабирование и распределение подзадач по вычислительным элементам	15
6.6.3. Программная реализация	15
6.6.4. Анализ эффективности	17
6.6.5. Результаты вычислительных экспериментов	20
6.7. Умножение матрицы на вектор при блочном разделении данных	23
6.7.1. Определение подзадач	23
6.7.2. Выделение информационных зависимостей	23
6.7.3. Масштабирование и распределение подзадач по вычислительным элементам	24

6.7.4.	Программная реализация	25
6.7.5.	Анализ эффективности	27
6.7.6.	Результаты вычислительных экспериментов	29
6.8.	Краткий обзор главы	33
6.9.	Обзор литературы	34
6.10.	Контрольные вопросы	34
6.11.	Задачи и упражнения	35
Глава 7.	Параллельные методы матричного умножения	1
7.1.	Постановка задачи	2
7.2.	Последовательный алгоритм	2
7.2.1.	Описание алгоритма	2
7.2.2.	Анализ эффективности	3
7.2.3.	Программная реализация	4
7.2.4.	Результаты вычислительных экспериментов	5
7.3.	Базовый параллельный алгоритм умножения матриц	7
7.3.1.	Определение подзадач	7
7.3.2.	Выделение информационных зависимостей	8
7.3.3.	Масштабирование и распределение подзадач	8
7.3.4.	Анализ эффективности	8
7.3.5.	Программная реализация	9
7.3.6.	Результаты вычислительных экспериментов	9
7.4.	Алгоритм умножения матриц, основанный на ленточном разделении данных	12
7.4.1.	Определение подзадач	12
7.4.2.	Выделение информационных зависимостей	12
7.4.3.	Масштабирование и распределение подзадач	13
7.4.4.	Анализ эффективности	13
7.4.5.	Программная реализация	14
7.4.6.	Результаты вычислительных экспериментов	15
7.5.	Блочный алгоритм умножения матриц	17
7.5.1.	Определение подзадач	17
7.5.2.	Выделение информационных зависимостей	18
7.5.3.	Масштабирование и распределение подзадач	19
7.5.4.	Анализ эффективности	19
7.5.5.	Программная реализация	20
7.5.6.	Результаты вычислительных экспериментов	21
7.6.	Блочный алгоритм, эффективно использующий кэш-память	22
7.6.1.	Последовательный алгоритм	23
7.6.2.	Параллельный алгоритм	24
7.6.3.	Результаты вычислительных экспериментов	25
7.7.	Краткий обзор главы	28
7.8.	Обзор литературы	29
7.9.	Контрольные вопросы	29

7.10.	Задачи и упражнения	30
Глава 8. Параллельные методы решения систем линейных уравнений		1
8.1.	Постановка задачи	1
8.2.	Метод Гаусса	2
8.2.1.	Общая схема метода	2
8.2.2.	Прямой ход метода Гаусса	3
8.2.3.	Обратный ход метода Гаусса	4
8.2.4.	Программная реализация	5
8.2.5.	Анализ эффективности	7
8.2.6.	Результаты вычислительных экспериментов	9
8.3.	Параллельный вариант метода Гаусса	10
8.3.1.	Выделение информационных зависимостей	10
8.3.2.	Масштабирование и распределение подзадач	10
8.3.3.	Программная реализация	11
8.3.4.	Анализ эффективности	14
8.3.5.	Результаты вычислительных экспериментов	15
8.4.	Метод сопряженных градиентов	18
8.4.1.	Последовательный алгоритм	18
8.4.2.	Организация параллельных вычислений	20
8.4.3.	Программная реализация	20
8.4.4.	Анализ эффективности	22
8.4.5.	Результаты вычислительных экспериментов	23
8.5.	Краткий обзор главы	26
8.6.	Обзор литературы	26
8.7.	Контрольные вопросы	27
8.8.	Задачи и упражнения	27
Глава 9. Сортировка данных		1
9.1.	Основы сортировки и принципы распараллеливания	2
9.2.	Пузырьковая сортировка	3
9.2.1.	Последовательный алгоритм пузырьковой сортировки	3
9.2.2.	Метод чет-нечетной перестановки	7
9.2.3.	Базовый параллельный алгоритм пузырьковой сортировки	7
9.2.4.	Блочный параллельный алгоритм пузырьковой сортировки	13
9.3.	Сортировка Шелла	20
9.3.1.	Последовательный алгоритм	20
9.3.2.	Организация параллельных вычислений	21
9.3.3.	Анализ эффективности	22
9.3.4.	Программная реализация	22
9.3.5.	Результаты вычислительных экспериментов	26
9.4.	Быстрая сортировка	29
9.4.1.	Последовательный алгоритм быстрой сортировки	29
9.4.2.	Параллельный алгоритм быстрой сортировки	31
9.4.3.	Обобщенный алгоритм быстрой сортировки	39

9.4.4. Сортировка с использованием регулярного набора образцов	44
9.5. Краткий обзор главы	52
9.6. Обзор литературы	54
9.7. Контрольные вопросы	54
9.8. Задачи и упражнения	54
Глава 10. Обработка графов	1
10.1. Задача поиска всех кратчайших путей	3
10.1.1. Последовательный алгоритм Флойда	3
10.1.2. Разделение вычислений на независимые части	4
10.1.3. Масштабирование и распределение подзадач по процессорам	4
10.1.4. Анализ эффективности параллельных вычислений	4
10.1.5. Программная реализация	5
10.1.6. Результаты вычислительных экспериментов	7
10.2. Задача нахождения минимального охватывающего дерева	10
10.2.1. Последовательный алгоритм Прима	11
10.2.2. Программная реализация последовательного алгоритма Прима	11
10.2.3. Разделение вычислений на независимые части	14
10.2.4. Анализ эффективности параллельных вычислений	14
10.2.5. Программная реализация параллельного алгоритма Прима	15
10.2.6. Результаты вычислительных экспериментов	17
10.3. Задача оптимального разделения графов	20
10.3.1. Постановка задачи оптимального разделения графов	21
10.3.2. Метод рекурсивного деления пополам	22
10.3.3. Геометрические методы	22
10.3.4. Комбинаторные методы	24
10.3.5. Сравнение алгоритмов разбиения графов	26
10.4. Краткий обзор главы	27
10.5. Обзор литературы	28
10.6. Контрольные вопросы	28
10.7. Задачи и упражнения	28
Глава 11. Решение дифференциальных уравнений в частных производных	1
11.1. Последовательные методы решения задачи Дирихле	2
11.2. Организация параллельных вычислений для систем с общей памятью	4
11.2.1. Использование OpenMP для организации параллелизма	4
11.2.2. Проблема синхронизации параллельных вычислений	5
11.2.3. Возможность неоднозначности вычислений в параллельных программах	8
11.2.4. Проблема взаимоблокировки	9
11.2.5. Исключение неоднозначности вычислений	10
11.2.6. Волновые схемы параллельных вычислений	12
11.2.7. Балансировка вычислительной нагрузки процессоров	17
11.3. Организация параллельных вычислений для систем с распределенной памятью	
11.3.1. Разделение данных	18

11.3.2. Обмен информацией между процессорами	19
11.3.3. Коллективные операции обмена информацией	22
11.3.4. Организация волны вычислений	23
11.3.5. Блочная схема разделения данных	24
11.3.6. Оценка трудоемкости операций передачи данных	27
11.4. Краткий обзор главы	28
11.5. Обзор литературы	29
11.6. Контрольные вопросы	29
11.7. Задачи и упражнения	29
<b>ГЛАВА 12. МНОГОЭКСТРЕМАЛЬНАЯ ОПТИМИЗАЦИЯ 1</b>	
12.1. ВВЕДЕНИЕ 1	
12.2. ПОСТАНОВКА ЗАДАЧИ 1	
12.3. МЕТОДЫ РЕШЕНИЯ ЗАДАЧ МНОГОЭКСТРЕМАЛЬНОЙ ОПТИМИЗАЦИИ 3	
12.4. РЕШЕНИЕ ОДНОМЕРНЫХ ЗАДАЧ	4
12.4.1. Индексный метод учета ограничений	4
12.4.2. Схема алгоритма	8
12.4.3. Программная реализация	10
12.4.4. Результаты численных экспериментов	14
12.5. РЕДУКЦИЯ РАЗМЕРНОСТИ ЗАДАЧИ	15
12.5.1. Использование отображений Пеано	15
12.5.2. Программная реализация	16
12.5.3. Результаты численных экспериментов	19
12.5.4. Способ построения развертки	20
12.6. ИСПОЛЬЗОВАНИЕ МНОЖЕСТВЕННЫХ ОТОБРАЖЕНИЙ	22
12.6.1. Основная схема	22
12.6.2. Программная реализация	23
12.6.3. Способ построения множественных отображений	23
12.7. ПАРАЛЛЕЛЬНЫЙ ИНДЕКСНЫЙ МЕТОД	25
12.7.1. Организация параллельных вычислений	25
12.7.2. Схема алгоритма	26
12.7.3. Результаты численных экспериментов	28
Глава 13. Программная система Паралаб для изучения и исследования методов параллельных вычислений 2	
13.1. Введение	2
13.2. Общая характеристика системы	3
13.3. Формирование модели вычислительной системы	5
13.3.1. Выбор топологии сети	5
13.3.2. Задание количества процессоров и ядер	7
13.3.3. Задание характеристик сети	8
13.4. Постановка вычислительной задачи и выбор параллельного метода решения	10
13.4.1. Умножение матрицы на вектор	12
13.4.2. Матричное умножение	14

13.4.3. Решение систем линейных уравнений	16
13.4.4. Сортировка данных	18
13.4.5. Обработка графов	21
13.4.6. Решение дифференциальных уравнений	24
13.4.7. Решение задач многоэкстремальной оптимизации	27
13.5. Определение графических форм наблюдения за процессом параллельных вычислений	29
13.5.1. Область "Выполнение эксперимента"	31
13.5.2. Область "Текущее состояние массива"	33
13.5.3. Область "Результат умножения матрицы на вектор"	33
13.5.4. Область "Результат умножения матриц"	34
13.5.5. Область "Результат решения системы уравнений"	34
13.5.6. Область "Результат обработки графа"	34
13.5.7. Область "Распределение тепла"	34
13.5.8. Область "Целевая функция"	35
13.5.9. Выбор процессора	35
13.6. Накопление и анализ результатов экспериментов	35
13.6.1. Общие результаты экспериментов	36
13.6.2. Просмотр итогов	36
13.7. Выполнение вычислительных экспериментов	38
13.7.1. Последовательное выполнение экспериментов	38
13.7.2. Выполнение экспериментов по шагам	39
13.7.3. Выполнение нескольких экспериментов	39
13.7.4. Выполнение серии экспериментов	41
13.7.5. Выполнение реальных вычислительных экспериментов	42
13.8. Использование результатов экспериментов: запоминание, печать и перенос в другие программы	43
13.8.1. Запоминание результатов	43
13.8.2. Печать результатов экспериментов	44
13.8.3. Копирование результатов в другие программы	44
Приложение: Таблица выполнимости методов на различных топологиях в системе Паралаб	45

Глава 1. Обзор архитектуры современных многоядерных процессоров	1
1.1. Параллелизм как основа высокопроизводительных вычислений	3
1.1.1. Симметрическая мультипроцессорность.	4
1.1.2. Одновременная многопотоковость.	6
1.1.3. Многоядерность.	7
1.2. Многоядерность – два, четыре, восемь – кто больше?	9
1.2.1. Процессоры Intel® Core™ и Intel® Xeon®.	9
1.2.2. Процессоры AMD Phenom™ и AMD Opteron™.	11
1.2.3. Процессоры IBM Power6.	12
1.2.4. Процессоры PowerXCell™ 8i.	13
1.2.5. Процессоры Sun UltraSPARC T1 и Sun UltraSPARC T2.	15
1.3. Ускорители вычислений	16
1.3.1. Ускоритель ClearSpeed™ Advance™ X620.	16
1.3.2. Настольный суперкомпьютер NVIDIA® Tesla™ D870.	17
1.4. Персональные мини-кластеры	19
1.4.1. Персональный суперкомпьютер T-Forge Mini.	19
1.4.2. Мини-кластер T-Edge Mini.	20
1.5. Краткий обзор раздела	20
1.6. Обзор литературы	21
1.7. Контрольные вопросы	21
1.8. Задачи и упражнения	22

## Глава 1. Обзор архитектуры современных многоядерных процессоров

«*Citius, Altius, Fortius*<sup>1</sup>» – девиз Олимпийских игр современности, как ни к какой другой области, применим к вычислительной технике. Воплощение в жизнь не раз видоизменявшего свою исходную формулировку, но до сих пор действующего эмпирического закона сформулированного в 1965 году Гордоном Муром, похоже, стало «делом чести» производителей аппаратного обеспечения. Из всех известных формулировок этого закона точку зрения потребителя/пользователя наилучшим образом отражает вариант: «производительность вычислительных систем удваивается каждый 18 месяцев». Мы сознательно не использовали термин «процессор», поскольку конечного пользователя вовсе не интересует, кто обеспечивает ему повышение мощности: процессор, ускоритель, видеокарта, – ему важен лишь сам факт роста возможностей «за те же деньги».

Правда, в последние несколько лет возможности увеличения мощности процессоров на основе повышения тактовой частоты оказались фактически исчерпаны, и производители, выбрав в качестве магистрального пути развития увеличение числа ядер на кристалле, были вынуждены призвать на помощь разработчиков программного обеспечения. Старые последовательные программы, способные использовать лишь одно ядро, теперь уже не будут работать быстрее на новом поколении процессоров «задаром» – требуется практически повсеместное внедрение программирования параллельного.

---

<sup>1</sup> «Быстрее, Выше, Сильнее» – лат.

Помимо представленной выше известна и другая формулировка закона Мура: «доступная (человечеству) вычислительная мощность удваивается каждые 18 месяцев». Зримое свидетельство этого варианта формулировки – список Top500 [106] самых высокопроизводительных вычислительных систем мира, обновляемый дважды в год. В 31-м списке Top500 (июнь 2008) впервые в истории был преодолен петафлопный порог производительности – суперкомпьютер Roadrunner [1012] производства компании IBM показал на тесте LINPACK 1,026 петафлопс (предыдущий «психологический» барьер в один терафлопс был преодолен системой ASCI Red [111] производства компании Intel в 1997 году – как видим, всего за 11 лет пик мощности вырос на три порядка). А суммарная мощность систем, представленных в 31-м списке Top500, составила 11,7 петафлопс. Много это или мало? Если взять за основу, что реальная производительность хорошей «персоналки» на четырехъядерном процессоре составляет порядка 20 гигафлопс, то весь список Top500 будет эквивалентен половине миллиона таких персоналок. Очевидно, что это лишь вершина айсберга. По данным аналитической компании Gartner общее число используемых в мире компьютеров превысило в 2008 году 1 миллиард.

Представленные в списке Top500 данные позволяют проследить характерные тенденции развития индустрии в сфере суперкомпьютерных вычислений. Первый список Top500 датирован июнем 1993 года и содержит 249 многопроцессорных систем с общей памятью и 97 суперкомпьютеров, построенных на основе единственного процессора; более 40% всех решений в нем были созданы на платформе, разработанной компанией Cray. Уже четырьмя годами позже в Top500 не осталось ни одного суперкомпьютера на основе единственного процессора, а взамен появилась первая система с производительностью всего в 10 гигафлопс (в 100 раз меньше, чем у лидера списка системы ASCI Red), относящаяся к довольно новому тогда кластерных вычислительных систем, которые сегодня занимают в Top500 80% списка и являются, по факту, основным способом построения суперкомпьютеров.

Основным преимуществом кластеров, предопределившим их повсеместное распространение, было и остается построение из стандартных массово выпускающихся компонент, как аппаратных, так и программных. Сегодня 75% систем в списке построены на основе процессоров компании Intel, чуть больше 13% – на процессорах компании IBM и 11% – компании AMD (на двух оставшихся производителей NEC и Cray приходится по одной системе соответственно); 81% систем используют всего два типа сетей передачи данных: Gigabit Ethernet или Infiniband; 85% систем работают под управлением операционной системы из семейства Linux. Как видим, разнообразием список не блещет, что является несомненным плюсом с точки зрения пользователей.

Однако для массового пользователя еще большим плюсом была бы возможность иметь персональный суперкомпьютер у себя на столе или, на худой конец, стоящий под столом. И кластера, принесшие в индустрию высокопроизводительных вычислений идею «собери суперкомпьютер своими руками», как нельзя лучше отвечают этой потребности. Сейчас трудно достоверно установить, какая система может быть названа первым в мире «персональным кластером», во всяком случае уже в начале 2001 года компания RenderCube [109] представила одноименный мини-кластер из 4-х двухпроцессорных систем, заключенных в кубический корпус со стороной всего в 42 см.

Тенденция «персонализации» супервычислений в последнее время развивается все активнее и недавно была подхвачена в том числе и производителями видеокарт, мощности которых возросли настолько, что возникло естественное желание использовать их не только в графических расчетах, но и в качестве ускорителей вычислений общего назначения. Соответствующие решения представлены в настоящее время компанией NVIDIA (семейство NVIDIA® Tesla™) и компанией AMD (семейство ATI FireStream™) и демонстрируют в силу специфики внутреннего устройства потрясающую (в сравнении с

универсальными процессорами) пиковую производительность, превышающую 1 терафлопс.

Данный раздел посвящен рассмотрению современных многоядерных процессоров, которые являются основой для построения самых быстродействующих вычислительных систем. Для полноты картины в разделе приводится также описание ряда аппаратных устройств (видеокарт и вычислительных сопроцессоров), которые могут быть использованы для существенного ускорения вычислений. И для завершения рассматриваемой темы в последней части раздела дается краткая характеристика «персональных» мини-кластеров, которые позволяют при достаточно «экономных» финансовых затратах приступить в решению имеющихся вычислительно-трудоемких задач с использованием высокопроизводительных вычислительных систем.

## 1.1. Параллелизм как основа высокопроизводительных вычислений

Без каких-либо особых преувеличений можно заявить, что все развитие компьютерных систем происходило и происходит под девизом «Скорость и быстрота вычислений». Если быстродействие первой вычислительной машины ENIAC составляло всего несколько тысяч операций в секунду, то самый быстрый на данный момент времени суперкомпьютер RoadRunner может выполнять уже квадриллионы ( $10^{15}$ ) команд. Темп развития вычислительной техники просто впечатляет – увеличение скорости вычислений в триллионы ( $10^{12}$ ) раз не многим более чем за 60 лет. Для лучшего понимания необычности столь стремительного развития средств ВТ часто приводят яркие аналогии – например, что если бы автомобильная промышленность развивалась с такой же динамикой, то сейчас бы автомобили весили бы порядка 200 грамм и тратили бы несколько литров бензина на миллионы километров!

История развития вычислительной техники представляет увлекательное описание замечательных научно-технических решений, радости побед и горечи поражений. Проблема создания высокопроизводительных вычислительных систем относится к числу наиболее сложных научно-технических задач современности и ее разрешение возможно только при всемерной концентрации усилий многих талантливых ученых и конструкторов, предполагает использование всех последних достижений науки и техники и требует значительных финансовых инвестиций. Важно отметить при этом, что при общем росте скорости вычислений в  $10^{12}$  раз, быстродействие самих технических средств вычислений увеличилось всего в несколько миллионов раз. И дополнительный эффект достигнут за счет введения *параллелизма* буквально на всех стадиях и этапах вычислений.

Не ставя целью в рамках данного материала подробное рассмотрение истории развития компьютерного параллелизма отметим, например, организацию независимости работы разных устройств ЭВМ (процессора и устройств ввода-вывода), появление многоуровневой памяти, совершенствование архитектуры процессоров (суперскалярность, конвейерность, динамическое планирование). Дополнительная информация по истории параллелизма, может быть получена, например, в [67]; здесь же выделим, как принципиально важный итог – многие возможные пути совершенствования процессоров практически исчерпаны (так, возможность дальнейшего повышения тактовой частоты процессоров ограничивается рядом сложных технических проблем) и наиболее перспективное направление на данный момент времени состоит в явной организации многопроцессорности вычислительных устройств.

Далее в разделе будут более подробно рассмотрены основные способы организации многопроцессорности – *симметричной мультипроцессорности* (*Symmetric Multiprocessor, SMP*), *одновременной многопотковости* (*Simultaneous Multithreading, SMT*) и *многоядерности* (*multicore*).

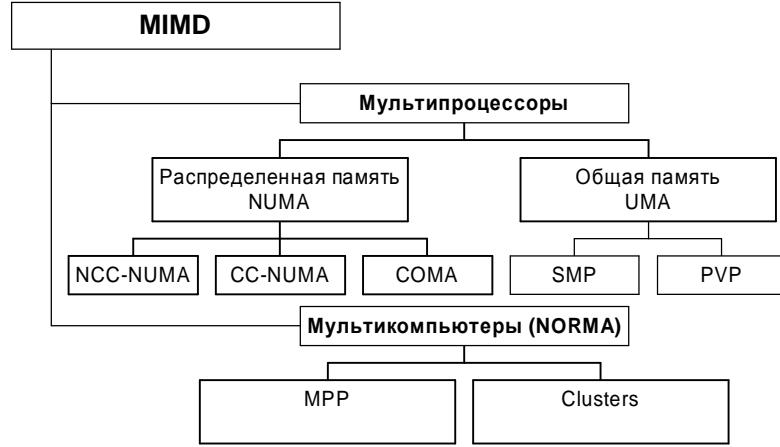


Рис. 1.1. Классификация многопроцессорных вычислительных систем

### 1.1.1. Симметрическая мультипроцессорность

Организация симметричной мультипроцессорности (*Symmetric Multiprocessor, SMP*), когда в рамках одного вычислительного устройства имеется несколько полностью равноправных процессоров, является практически первым использованным подходом для обеспечения многопроцессорности – первые вычислительные системы такого типа стали появляться в середине 50-х – начале 60-х годов, однако массовое применение SMP систем началось только в середине 90-х годов.

Следует отметить, что SMP системы входят в группу MIMD (*multi instruction multi data*) вычислительных систем в соответствии с классификацией Флинна. Поскольку эта классификация приводит к тому, что практически все виды параллельных систем (несмотря на их существенную разнородность) относятся к одной группе MIMD, для дальнейшей детализации класса MIMD предложена практически общепризнанная структурная схема (см. [47,99]) - см. рис. 1.1. В рамках данной схемы дальнейшее разделение типов многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах. Данный поход позволяет различать два важных типа многопроцессорных систем – *multiprocessors* (мультипроцессоры или системы с общей разделяемой памятью) и *multicomputers* (мультикомпьютеры или системы с распределенной памятью).

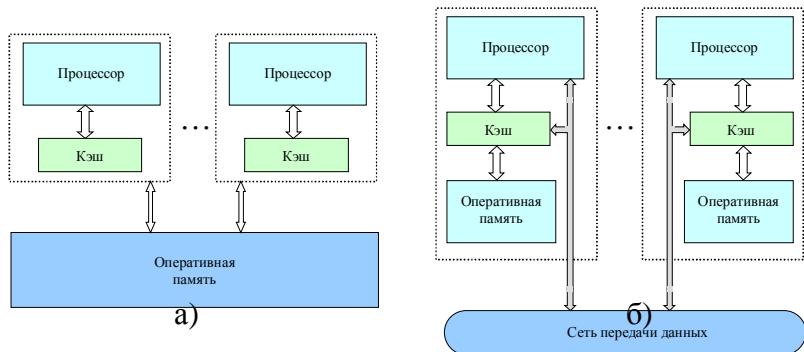


Рис. 1.2. Архитектура многопроцессорных систем с общей (разделяемой) памятью: системы с (а) однородным и (б) неоднородным доступом к памяти

Для дальнейшей систематики мультипроцессоров учитывается способ построения общей памяти. Возможный подход – использование единой (централизованной) *общей памяти* (*shared memory*) – см. рис. 1.2. Такой подход обеспечивает *однородный доступ к памяти* (*uniform memory access* or *UMA*) и служит основой для построения *векторных параллельных процессоров* (*parallel vector processor* or *PVP*) и *симметричных мультипроцессоров* (*symmetric multiprocessor* or *SMP*). Среди примеров первой группы суперкомпьютер Cray T90, ко второй группе относятся IBM eServer, Sun StarFire, HP Superdome, SGI Origin и др.

Одной из основных проблем, которые возникают при организации параллельных вычислений на такого типа системах, является доступ с разных процессоров к общим данным и обеспечение, в этой связи, *однозначности (когерентности) содержимого разных кэшей* (*cache coherence problem*). Дело в том, что при наличии общих данных копии значений одних и тех же переменных могут оказаться в кэшах разных процессоров. Если в такой ситуации (при наличии копий общих данных) один из процессоров выполнит изменение значения разделяемой переменной, то значения копий в кэшах других процессорах окажутся не соответствующими действительности и их использование приведет к некорректности вычислений. Обеспечение однозначности кэшей обычно реализуется на аппаратном уровне – для этого после изменения значения общей переменной все копии этой переменной в кэшах отмечаются как недействительные и последующий доступ к переменной потребует обязательного обращения к основной памяти. Следует отметить, что необходимость обеспечения когерентности приводит к некоторому снижению скорости вычислений и затрудняет создание систем с достаточно большим количеством процессоров.

Наличие общих данных при выполнении параллельных вычислений приводит к необходимости *синхронизации взаимодействия* одновременно выполняемых потоков команд. Так, например, если изменение общих данных требует для своего выполнения некоторой последовательности действий, то необходимо обеспечить *взаимоисключение* (*mutual exclusion*) с тем, чтобы эти изменения в любой момент времени мог выполнять только один командный поток. Задачи взаимоисключения и синхронизации относятся к числу классических проблем, и их рассмотрение при разработке параллельных программ является одним из основных вопросов параллельного программирования.

Общий доступ к данным может быть обеспечен и при физически распределенной памяти (при этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти) – см. рис. 1.2. Такой подход именуется как *неоднородный доступ к памяти* (*non-uniform memory access* or *NUMA*). Среди систем с таким типом памяти выделяют:

- Системы, в которых для представления данных используется только локальная кэш-память имеющихся процессоров (*cache-only memory architecture* or *COMA*); примерами таких систем являются, например, KSR-1 и DDM;
- Системы, в которых обеспечивается когерентность локальных кэшей разных процессоров (*cache-coherent NUMA* or *CC-NUMA*); среди систем данного типа SGI Origin 2000, Sun HPC 10000, IBM/Sequent NUMA-Q 2000;
- Системы, в которых обеспечивается общий доступ к локальной памяти разных процессоров без поддержки на аппаратном уровне когерентности кэша (*non-cache coherent NUMA* or *NCC-NUMA*); к данному типу относится, например, система Cray T3E.

Использование распределенной общей памяти (*distributed shared memory* or *DSM*) упрощает проблемы создания мультипроцессоров (известны примеры систем с несколькими тысячами процессоров), однако, возникающие при этом проблемы эффективного использования распределенной памяти (время доступа к локальной и

удаленной памяти может различаться на несколько порядков) приводят к существенному повышению сложности параллельного программирования.

### 1.1.2. Одновременная многопотоковость

Организация симметричной мультипроцессорности позволяет достаточно легко увеличивать производительность вычислительных устройств. Однако такое решение обладает плохой масштабируемостью при увеличении числа процессоров из-за проблем с обеспечением когерентности кэш-памяти разных процессоров – SMP системы содержат, как правило 2 или 4, реже – 8, и совсем редко большее количество процессоров. Кроме того, такой подход является сравнительно дорогим решением.

С другой стороны, проанализировав эффективность современных сложных процессоров, насчитывающих в своем составе десятки и сотни миллионов транзисторов, можно обратить внимание, что при выполнении большинства операций не все составные компоненты процессоров оказываются полностью задействованными (по имеющимся оценкам, средняя загрузка процессора составляет всего лишь порядка 30%). Так, если в данный момент времени выполняется операция целочисленной арифметики, блок процессора для выполнения вещественной операции окажется простаивающим. Для повышения загрузки процессора можно организовать *спекулятивное (опережающее) исполнение* операций, однако воплощение такого подхода требует существенного усложнения логики аппаратной реализации процессора. Гораздо проще было бы, если в программе заранее были бы предусмотрены последовательности команд (*потоки*), которые могли бы быть выполнены параллельно и независимо друг от друга. Такой подход тем более является целесообразным, поскольку поддержка многопоточного исполнения может быть обеспечена и на аппаратном уровне за счет соответствующего расширения возможностей процессора (и такая доработка является сравнительно простой).

Данная идея поддержки *одновременной многопоточности* (*simultaneous multithreading, SMT*) была предложена в 1995 году в университете Вашингтона Дином Тулсеном (Dean Tullsen) и позднее активно развита компанией Интел под названием технологии *гиперпоточности* (*hyper threading, HT*). В рамках такого подхода процессор дополняется средствами запоминания состояния потоков, схемами контроля одновременного выполнения нескольких потоков и т.д. За счет этих дополнительных средств на активной стадии выполнения может находиться несколько потоков; при этом одновременно выполняемые потоки конкурируют за исполнительные блоки единственного процессора и, как результат, выполнение отдельных потоков может блокироваться, если требуемые в данный момент времени блоки процессора оказываются уже задействованными. Как правило, число аппаратно-поддерживаемых потоков равно 2, в более редких случаях этот показатель достигает 4 и даже 8. Важно при этом подчеркнуть, что аппаратно-поддерживаемые потоки на логическом уровне операционных систем Linux и Windows воспринимаются как отдельные процессоры, т.е., например, единственный процессор с двумя аппаратно-поддерживаемыми потоками в менеджере Task Manager операционной системы Windows диагностируется как два отдельных процессора.

Использование процессоров с поддержкой многопотоковости может приводить к ускорению вычислений (важно отметить – при надлежащей реализации программ). Так, имеется большое количество демонстраций, показывающих, что на процессорах компаний Интел с поддержкой технологии гиперпоточности достигается повышение скорости вычислений порядка 30%.

### 1.1.3. Многоядерность

Технология одновременной многопоточности позволяет достичь многопроцессорности на логическом уровне. Еще раз отметим, затраты на поддержку такой технологии являются сравнительно небольшими, но и получаемый результат достаточно далек от максимально-возможного – ускорение вычислений от использования многопоточности оказывается равным порядка 30%. Дальнейшее повышение быстродействия вычислений при таком подходе по прежнему лежит на пути совершенствования процессора, что – как было отмечено ранее – требует разрешения ряда сложных технологических проблем.

Возможное продвижение по направлению к большей вычислительной производительности может быть обеспечено на основе «парадоксального», на первый взгляд, решения – возврат к более «простым» процессорам с более низкой тактовой частотой и с менее сложной логикой реализации! Такой неординарный ход приводит к тому, что процессоры становятся менее энергоемкими<sup>2)</sup>, более простыми для изготовлениями и, как результат, более надежными. А также – что является чрезвычайно важным – «простые» процессоры требуют для своего изготовления меньшее количество логических схем, что приводит к освобождению в рамках кремниевого кристалла, используемого для изготовления процессоров, большого количества свободных транзисторов. Эти свободные транзисторы, в свою очередь, могут быть использованы для реализации дополнительных вычислительных устройств, которые могут быть добавлены к процессору. Фактически, данный подход позволяет реализовать в единственном кремниевом кристалле несколько *вычислительных ядер* в составе одного многоядерного процессора, при этом по своим вычислительным возможностям эти ядра могут не уступать обычным (одноядерным) процессорам. Поясним сказанное на примере рис. 1.3. В центре рисунка (см. рис. 1.3б) приведены показатели исходного процессора, принятые за 1 для последующего сравнения. Пусть для повышения быстродействия процессора его тактовая частота увеличена на 20% (см. 1.3а), тогда производительность процессора увеличится – только не на 20%, а, например, на 13% (приводимые здесь числовые значения имеют качественный характер), в то время, как энергопотребление потребления возрастет существенно – пусть для примера на 73%. Данный пример является, на самом деле, очень характерным – увеличение тактовой частоты процессора приводит в большинстве случаев к значительному росту энергопотребления. Теперь уменьшим тактовую частоту процессора – опять же на 20% (см. рис. 1.3в). В результате снижения тактовой частоты производительность процессора, конечно, уменьшится, но, опять же, не на 20%, а примерно на ту же величину 13% (т.е. станет равной 87% от производительности исходного процессора). И опять же, энергопотребление процессора уменьшится, причем достаточно значительно – до уровня порядка 51% энергопотребления исходного процессора. И тогда, добавив в процессор второе вычислительное ядро за счет появившихся свободных транзисторов, мы можем довести суммарные показатели процессора по энергопотреблению до уровня 1.02 энергопотребления исходного процессора, а производительность – до уровня 1.73 !!!

---

<sup>2)</sup> Проблема энергопотребления является одной из наиболее сложных для процессоров с высокой тактовой частотой

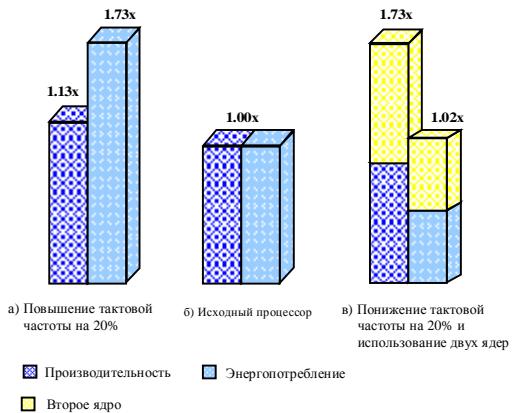


Рис. 1.3. Демонстрация зависимости между тактовой частотой, энергопотреблением и производительностью процессора

На логическом уровне архитектура многоядерного процессора соответствует практически архитектуре симметричного мультипроцессора (см. рис. 1.2 и рис. 1.4). На рис. 1.4 приведена возможная архитектура двухядерного процессора – различия для разных многоядерных процессоров могут состоять в количестве имеющихся ядер и в способах использования кэш-памяти ядрами процессора – кэш-память может быть как и общей, так и распределенной для разных ядер. Так, на рис. 1.4 кэш-память первого уровня L1 локальна для каждого ядра, в то же время кэш-память всех последующих уровней и оперативная память является общей.

Как следует из проведенного рассмотрения, многоядерность позволяет повышать производительность процессоров и данный подход обладает целым рядом привлекательных моментов (уменьшение энергопотребления, снижение сложности логики процессоров и т.п.). Все сказанное приводит к тому, что многоядерность становится одной из основных направлений развития компьютерной техники. Значимость такого подхода привела даже к тому, что известный закон Мура теперь формулируется в виде «Количество вычислительных ядер в процессоре будет удваиваться каждые 18 месяцев». В данный момент для массового использования доступны двух- и четырех- ядерные процессоры, компании-разработчики объявили о подготовке шести-ядерных процессоров. В научно-технической литературе наряду с рассмотрением обычных многоядерных (*multi-core*) процессоров начато широкое обсуждение процессоров с массовой многоядерностью (*many-core*), когда в составе процессоров будут находиться сотни и тысячи ядер!

И в заключение следует отметить еще один принципиальный момент – потенциал производительности многоядерных процессоров может быть задействован только при надлежащей разработке программного обеспечения – программы должны быть очень хорошо распараллелены. А, как известно, сложность разработки параллельных программ значительно превышает трудоемкость обычного последовательного программирования. И, тем самым, проблема обеспечения высокопроизводительных вычислений перемещается теперь из области компьютерного оборудования в сферу параллельного программирования. И здесь нужны новые идеи и перспективные технологии для организации массового производства параллельных программ.

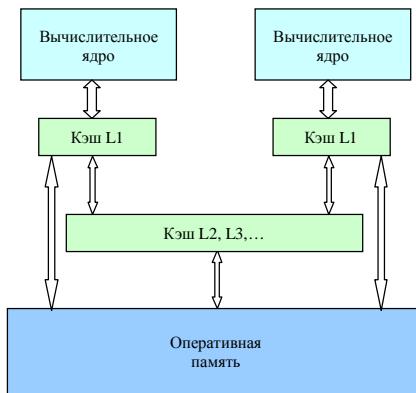


Рис. 1.4. Архитектура двуядерного процессора

## 1.2. Многоядерность – два, четыре, восемь – кто больше?

Дополним теперь общее рассмотрение многоядерного направления развития компьютерной техники характеристикой ряда конкретных широко применяемых в данный момент времени многоядерных процессоров основных компаний-разработчиков Intel, AMD, IBM, Sun.

### 1.2.1. Процессоры Intel® Core™ и Intel® Xeon®

Как просто было когда-то сравнивать процессоры компании Intel между собой. Все знали, есть Pentium, есть его «урезанный» вариант Celeron, а в остальном, чем выше частота, тем лучше. Эта простота была следствием того факта, что в формуле, определяющей производительность вычислительной системы «тактовая частота процессора × число инструкций, выполняемых за один такт (Instructions Per Cycle, IPC)» переменной величиной была только частота. Необходимо, конечно, отметить, что получаемая по этой формуле величина, дает только так называемую «пиковую производительность», приблизиться к которой на практике можно лишь на отдельных специально подобранных задачах. Именно поэтому сравнение вычислительных систем, в том числе в списке Top500 и аналогичных, выполняется на основе производительности, показанной на стандартном тесте, в качестве которого повсеместно используется LINPACK [110]. Как только наращивание тактовой частоты прекратилось, компании Intel понадобился другой способ описания и градации выпускаемых процессоров. Сегодня все процессоры производства Intel делятся, прежде всего, по назначению: для настольных систем, ноутбуков, серверов и рабочих станций и т.д. Затем в каждом классе выделяют серии процессоров, отличающиеся между собой по некоторым ключевым характеристикам. В классе настольных и мобильных систем сегодня «царствуют» представители семейства Intel® Core™2, в серверном сегменте – процессоры Intel® Xeon®, при этом и те и другие построены на микроархитектуре Intel® Core™, пришедшей в 2006 году на смену архитектуре Intel® NetBurst™.

Особенностям микроархитектуры Intel® Core™, позволившей компании Intel существенно потеснить своего основного конкурента - компанию AMD, посвящено множество материалов и публикаций. Не ставя перед собой задачу подробного ее обсуждения, кратко отметим лишь ключевые моменты, выделяемые самими разработчиками.



Рис. 1.5. Архитектура Intel® Core™

- Wide Dynamic Execution. Если основой повышения производительности процессоров архитектуры NetBurst была тактовая частота, то в архитектуре Core на первое место вышло число инструкций за такт (с учетом увеличения этого показателя за счет наращивания числа ядер): IPC каждого ядра в этой архитектуре равно 4, таким образом пиковая производительность четырехъядерных процессоров, например, равна « $16 \times$  на тактовую частоту».

- Advanced Smart Cache. Кэш второго уровня в архитектуре Core является общим на каждую пару ядер (четырехъядерные процессоры Intel сегодня фактически представляют собой два двухъядерных, размещенных на одном кристалле), что позволяет как динамически менять его «емкость» для каждого ядра из пары, так и использовать преимущества совместного использования ядрами данных, находящихся в кэше. Кроме того, в случае активного использования всего одного ядра, оно «задаром» получает кэш вдвое большего размера, чем было бы в случае отдельного кэша второго уровня на каждое ядро.

- Advanced Digital Media Boost. По сравнению с NetBurst в архитектуре Core была значительно улучшена работа с векторными расширениями SSE. С точки зрения конечного пользователя основным из этих улучшений, помимо добавления новых команд, стала способность процессоров выполнять SSE-инструкции за один такт вместо двух в NetBurst.

- Intelligent Power Capacity. Процессоры на архитектуре Core получили возможность как интерактивного отключения незадействованных в данный момент подсистем, так и «динамического» понижения частоты ядер, что дало возможность существенно снизить тепловыделение (Thermal Design Power, TDP), что особенно положительно сказалось на процессорах для настольных и мобильных систем. Так, двухъядерный Pentium D с частотой 2,8 ГГц имел TDP 130 Вт, тогда как четырехъядерный Core 2 Quad Q9300 с частотой 2,5 ГГц – всего 95 Вт.

Кроме того необходимо отметить существенно уменьшившийся по сравнению с 31-стадийным в последних процессорах архитектуры NetBurst конвейер – его длина в архитектуре Core составляет 14 стадий, плюс «честную» 64-разрядность, плюс в очередной раз доработанное предсказание ветвлений, плюс многое, оставшееся за кадром...

Приведем технические данные текущих лидеров в классе настольных и серверных процессоров.

### **Процессор Intel® Core™2 Quad Q9650**

Тактовая частота: 3 ГГц.

Число ядер: 4.

Кэш второго уровня: 12 Мб (по 6 Мб на каждую пару ядер).

Частота системной шины: 1333 МГц.

Технологический процесс: 45 нанометров.

### **Процессор Intel® Xeon® X7460**

Тактовая частота: 2,66 ГГц.

Число ядер: 6.

Кэш второго уровня: 9 Мб (по 3 Мб на каждую пару ядер).

Кэш третьего уровня: 16 Мб.

Частота системной шины: 1066 МГц.

Технологический процесс: 45 нанометров.

В заключение отметим еще один весьма важный факт – помимо пиковой производительности той или иной архитектуры и соответственно процессоров, построенных на ее основе, значимым обстоятельством для конечного потребителя является процент мощности, который можно «отжать от пика». Для систем в Top500, построенных на процессорах компании Intel, этот показатель составляет в 31-м списке 61%, при этом «удельная мощность» в расчете на один процессор/ядро равна 6,23 гигафлопс (необходимо заметить, конечно, что значительная часть этих систем введена в строй уже несколько лет назад и построена не на новейших процессорах).

### 1.2.2. Процессоры AMD Phenom™ и AMD Opteron™

Компания AMD основана в 1969 году (всего на год позже, чем Intel) и в сознании рядового пользователя прочно занимает место главного конкурента Intel на рынке процессоров для настольных систем и отчасти на рынке серверных, практически всегда при этом выступая в роли догоняющего. Если принимать во внимание только «внешние» факторы, вроде рыночной доли, то ситуация, действительно, может быть воспринята именно так. И в этом свете основной успех компании за последнее десятилетие связан с выпуском в 2003 году 64-битных процессоров AMD Opteron™, быстро завоевавших популярность и позволивших AMD значительно упрочить свое положение, в том числе в сегменте высокопроизводительных решений. Достаточно отметить, что в 28-м списке Top500 (ноябрь 2006) доля систем, построенных на основе процессоров AMD, достигла своего исторического максимума и составила 22,6%, против 52,6% у компании Intel и 18% у компании IBM. Однако кроме такого чисто количественного сравнения, в котором AMD неизменно проигрывает своим конкурентам, есть еще показатели качественные, и тут компания нередко за прошедшие годы бывала первопроходцем и реализовывала действительно интересные архитектурные решения.

Среди прочего это и интеграция в процессор северного моста, что дает более быстрый доступ к оперативной памяти и использование Direct Connect Architecture для взаимодействия процессоров между собой посредством высокоскоростной шины HyperTransport™, позволяющей без существенных потерь в производительности объединять в рамках одной системы до 8 процессоров Opteron. Кроме того нужно отметить, что в процессорах Opteron реализована «честная» четырехъядерность (Native Quad-Core Design), двухпотоковое управление 128-битными SSE-инструкциями, выполнение до четырех операций с плавающей точкой двойной точности за такт, расширенная технология оптимизации энергопотребления (Enhanced AMD PowerNow!) и многое другое. Последними серверными процессорами компании AMD являются четырехъядерные модели Opteron 3G на ядре Barcelona.

На рынке настольных систем основное оружие компании AMD сегодня – процессоры AMD Phenom™. Процессоры Phenom построены на той же микроархитектуре (AMD K10), что и серверные Opteron. Помимо уже отмеченных особенностей можно упомянуть наличие в процессорах Phenom кэша третьего уровня, пиковую пропускную способность шины HyperTransport до 16 Гб/с, поддержку 128-битных операций SSE, работу кэша второго уровня на частоте ядра, технологию улучшенной защиты от вирусов (NX бит / Enhanced Virus Protection). Текущее поколение процессоров Phenom выпускается по технологии 65 нм.

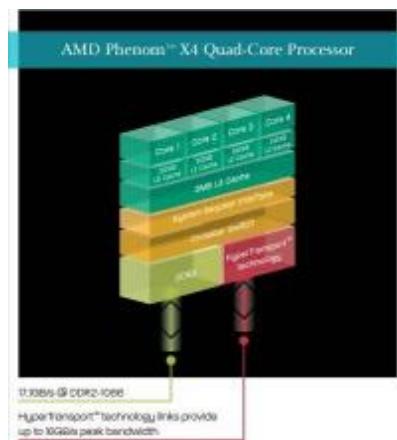


Рис. 1.6. Архитектура AMD Phenom™

Приведем технические данные текущих лидеров в классе настольных и серверных процессоров.

#### Процессор AMD Phenom™ X4 Quad-Core 9950

Тактовая частота: 2,6 ГГц.

Число ядер: 4.

Кэш второго уровня: 2 Мб (по 512 Кб на каждое ядро).

Кэш третьего уровня: 2 Мб (общий на все ядра).

Частота интегрированного контроллера памяти: 1066 МГц.

Технологический процесс: 65 нанометров.

#### Процессор Third-Generation AMD Opteron™ 8360 SE

Тактовая частота: 2,5 ГГц.

Число ядер: 4.

Кэш второго уровня: 2 Мб (по 512 Кб на каждое ядро).

Кэш третьего уровня: 2 Мб (общий на все ядра).

Частота интегрированного контроллера памяти: 2000 МГц.

Технологический процесс: 65 нанометров.

В заключение, как и для процессоров компании Intel, приведем усредненные данные из списка Top500. Для систем в Top500, построенных на процессорах компании AMD, отношение «показанная мощность/пиковая мощность» составляет в 31-м списке 71%, при этом «удельная мощность» в расчете на один процессор/ядро равна 4,48 гигафлопс. Как и ранее отметим, что значительная часть этих систем построена не на новейших процессорах.

#### 1.2.3. Процессоры IBM Power6

История компании IBM значительно длиннее, чем у Intel и AMD, и, в отличие от последних, IBM никогда не производила только процессоры. Фактически, компания, говоря сегодняшним языком, всегда пыталась поставлять «готовые решения». Однако обсуждения всего списка продукции IBM выходит за рамки данного материала, и мы остановимся только на процессорах, которые выпускает компания сегодня, и на основе которых строят как сервера «начального» уровня, так и суперкомпьютеры вроде Roadrunner или BlueGene.

Микропроцессорная архитектура Power (расшифровывается как Performance Optimization With Enhanced RISC) имеет не менее богатую историю, чем сама компания IBM. Начиная с 1990 года, когда были выпущены первые компьютеры на основе процессоров Power, и по сегодняшний день архитектура постоянно развивается, с каждым поколением процессоров привнося значительные новшества. Текущая версия процессоров Power – Power6 выпущена в середине 2007 года, тем не менее, уже 7 систем в 31-м списке Top500 построено на основе этих процессоров.

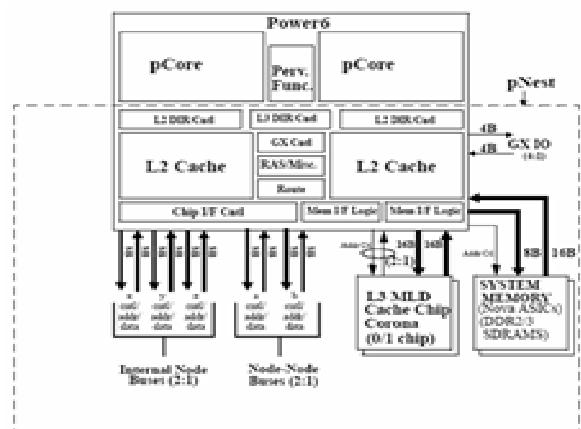


Рис. 1.7. Процессор IBM Power6

Процессор Power6 выпускается по 65 нм технологическому процессу. Максимальная частота серийно выпускаемых образцов на сегодня равна 4,7 ГГц.

Процессоры Power6 имеют два ядра, способных выполнять по два потока команд одновременно, по 4 Мб кэша второго уровня на каждое ядро, 32 Мб кэша третьего уровня на отдельном кристалле, присоединенного к шине с пропускной способностью 80 Гб/с. Каждое ядро содержит по два блока работы с целыми числами и числами с плавающей точкой соответственно. Однако главной изюминкой процессора Power6 является блок десятичных вычислений с плавающей точкой, аппаратно реализующий более 50 команд для выполнения математических операций над вещественными числами в десятичном представлении и перевода из двоичной системы счисления в десятичную и обратно. Второе не менее важное отличие Power6 от процессоров предыдущих серий – отказ IBM от внеочередного (out-of-order) исполнения команд, что стало одним из основных факторов, позволившим поднять частоту процессоров выше 4 ГГц.

Процессоры Power6 поставляются в многочиповом корпусе, аналогично Power5, вмещающем до 4 процессоров и общий кэш третьего уровня. В многопроцессорной конфигурации возможна «связка» из 32 процессоров посредством двух шин межпроцессорного обмена с пропускной способностью 50 Гб/с.

В заключение, как и ранее, приведем усредненные данные из списка Top500. Для систем в Top500, построенных на процессорах IBM семейства Power, отношение «показанная мощность/пиковая мощность» составляет в 31-м списке 77%, при этом «удельная мощность» в расчете на один процессор/ядро равна 3,71 гигафлопс. Как и ранее отметим, что значительная часть этих систем построена не на новейших процессорах.

Если же брать в расчет только системы на процессорах Power6, то картина несколько меняется: отношение «показанная мощность/пиковая мощность» для этих систем составляет 65%, а «удельная» на одно ядро – 12,14 гигафлопс. Существенно меньший показатель по показанной мощности не в последнюю очередь объясняется отказом от внеочередного исполнения команд, затрудняющего и без того непростую задачу достижения пиковой производительности. Что касается «удельной мощности», то здесь с наилучшей стороны проявляет себя высокая тактовая частота процессоров Power6.

#### 1.2.4. Процессоры PowerXCell™ 8i

Рассказ о процессоре PowerXCell™ 8i начать нужно, конечно же, с его прямого предка – процессора Cell, разработанного альянсом STI (Sony, Toshiba, IBM) в первую очередь для использования в игровых приставках Sony PlayStation 3. В процессе создания этого процессора были приняты весьма интересные решения, дающие в итоге очень высокую пиковую производительность (более 200 гигафлопс, правда, только для вещественной арифметики одинарной точности), но требующие в качестве платы более сложного программирования.

Прежде всего отметим, что процессор Cell имеет существенно «неоднородное» устройство. Он состоит из одного двухъядерного Power Processor Element (PPE) и 8 Synergistic Processor Element (SPE). PPE построен на архитектуре PowerPC, и «отвечает» в процессоре Cell за исполнение кода общего назначения (операционной системы в частности), а также контролирует работу потоков на сопроцессорах SPE. Ядра PPE 64-разрядны и, также как и Power6, используют поочередный (in-order) порядок исполнения команд. PPE имеет блок векторных операций Vector Multimedia eXtensions (VMX), кэш первого уровня размеров 64 Кб (по 32 Кб на кэш инструкций и данных) и кэш второго уровня размером 512 Кб.

В отличие от PPE SPE-ядра представляют собой специализированные векторные процессоры, ориентированные на быструю потоковую работу с SIMD-инструкциями. Архитектура SPE довольно проста: четыре блока для работы с целочисленными векторными операциями и четыре блока для работы с числами с плавающей запятой. Большинство арифметических инструкций представляют данные в виде 128-разрядных векторов, разделённых на четыре 32-битных элемента. Каждый SPE оснащён 128 регистрами, разрядность которых – 128-бит. Вместо кэша первого уровня SPE содержит 256 Кб собственной «локальной памяти» (local memory, также называемой local store или LS) разделённой на четыре отдельных сегмента по 64 Кб каждый, а также DMA-контроллер, который предназначен для обмена данными между основной памятью (RAM) и локальной памятью SPE (LS), минуя PPE. Доступ к LS составляет 6 тактов, что больше, чем время обращения к кэшу первого уровня, но меньше, чем к кэшу второго для большинства современных процессоров. SPE-ядра, также как и PPE, используют упорядоченную схему (in-order) исполнение инструкций.

Частота всех ядер в процессоре Cell составляет 3,2 ГГц, что дает производительность одного SPE в  $3,2 \times 4 \times 2 = 25,6$  гигафлопс (последняя двойка в произведении за счет двух конвейеров, позволяющих за один такт выполнять операции умножения и сложения над вещественными числами). Таким образом, пиковая производительность всего процессора Cell получается превышающей 200 гигафлопс.

Модель программирования под процессор Cell «изначально» многопоточная, поскольку на SPE могут выполняться только специализированные потоки. Данные, с которыми они работают, должны располагаться в LS, соответственно типичным подходом является их предвыборка. В целом Cell весьма эффективно справляется с «потоковой» обработкой, характерной для мультимедиа, для задач кодирования, сжатия и т.д.

Основное отличие процессора PowerXCell™ 8i от своего «предка» состоит в значительном улучшении работы с вещественными числами двойной точности, что позволило довести пиковую производительность на них до уровня в 100 гигафлопс. Кроме того PowerXCell™ 8i производится по 65 нм технологии, в отличие от 90 нм, использующихся в Cell. Наконец, в PowerXCell™ 8i был кардинально (до 32 Гб) увеличен объем поддерживаемой памяти.

В настоящий момент в Top500 три системы построены на процессорах PowerXCell 8i, в том числе лидер списка. Как и ранее, приведем усредненные данные из списка Top500 по этим трем системам. Отношение «показанная мощность/пиковая мощность» систем на основе процессоров PowerXCell 8i составляет в 31-м списке 74%, при этом «удельная мощность» в расчете на одно ядро равна 8,36 гигафлопс (то есть порядка 80 гигафлопс на процессор).

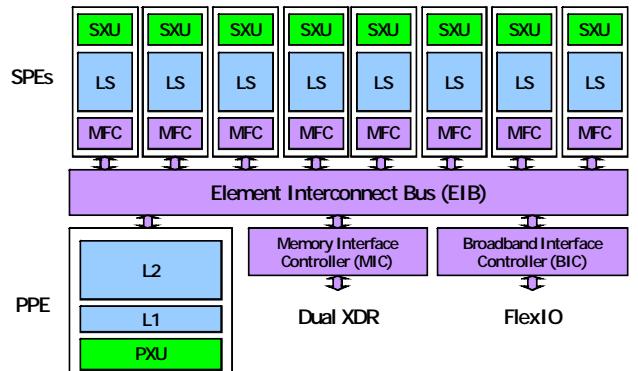


Рис. 1.8. Процессор Cell

### 1.2.5. Процессоры Sun UltraSPARC T1 и Sun UltraSPARC T2

Начиная разработку микроархитектуры UltraSPARC Architecture, компания Sun Microsystems подошла к процессу с позиций, существенно отличающихся от остальных производителей. В многоядерных процессорах Intel, AMD и IBM каждое ядро фактически является полноценным исполнительным устройством, ориентированным на выполнение кода общего назначения, а в процессорах семейства Cell SPE-ядра, напротив, в принципе не могут исполнять такой код и, по сути, являются сопроцессорами. В основу процессоров UltraSPARC T1 (кодовое имя Niagara), выпущенных на рынок в 2005 году и UltraSPARC T2 (кодовое имя Niagara-2), выпущенных в 2007, положена идея «многопоточности» для достижения высокой производительности не путем ускорения выполнения одного потока команд, а за счет обработки большого числа потоков в единицу времени. Как результат процессоры UltraSPARC T1 способны выполнять 32 потока одновременно (на восьми «четырехпоточных» ядрах), а процессоры UltraSPARC T2 – 64 потока (на восьми «восьмипоточных» ядрах). Эта многопоточность аппаратная (как, например, HyperThreading у компании Intel), то есть операционная система воспринимает UltraSPARC T1 и UltraSPARC T2 как 32 и 64 процессора соответственно.

В обоих процессорах компания Sun реализовала технологию, названную ими CoolThreads, позволяющую значительно снизить энергопотребление – TDP процессоров UltraSPARC T1 не превышает 79 Вт (по 2,5 ватта на поток), процессоров UltraSPARC T2 – 123 Вт (всего 2 ватта на поток).

Технические характеристики процессора Sun UltraSPARC T1:

- Тактовая частота: 1,0 или 1,2 ГГц.
- Число ядер: 8 (по 4 потока на каждое).
- Кэш инструкций первого уровня: 16 Кб на каждое ядро.
- Кэш данных первого уровня: 8 Кб на каждое ядро.
- Кэш второго уровня: 3 Мб (общий на все ядра).
- Интерфейс JBUS с пиковой пропускной способностью 3,1 Гб/с, 128-битной шиной частотой от 150 до 200 МГц.
- 90 нм технологический процесс.
- Энергопотребление: 72 Вт, пиковое – 79 Вт.

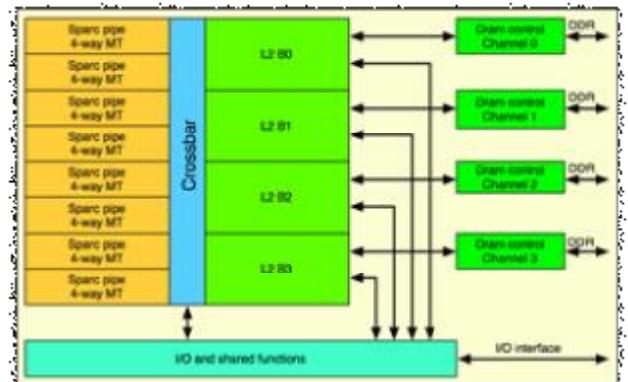


Рис. 1.9. Процессор Sun UltraSPARC T1

Переключение между потоками в процессоре UltraSPARC T1 осуществляется по циклической схеме на каждом такте, т.е. в каждый конкретный момент времени активен только один из четырех потоков ядра. Однако в случае, если в потоке возникает простой (например, при кэш-промахе), ядро переключается на работу с другим потоком. Такая стратегия позволяет скрыть возникающие задержки доступа к памяти при наличии достаточного количества потоков исполнения. Ядра UltraSPARC T1 по функциональности аналогичны процессорам предыдущего поколения UltraSPARC III, но существенно упрощены архитектурно, например, сокращены возможности прогноза ветвлений и спекулятивного выполнения команд, а число стадий конвейера уменьшено до шести (14 в UltraSPARC III).

Интересная особенность процессоров UltraSPARC T1 и T2 – наличие встроенного в ядро криптографического модуля (сопроцессора), реализующего на аппаратном уровне

алгоритм RSA с 2048-разрядными ключами. Сопроцессор ядер в UltraSPARC T2 дополнительно поддерживает алгоритмы шифрования DES, 3DES, RC4, AES, SHA, MD5, CRC, а также алгоритм генерации случайных чисел.

Основной недостаток процессора UltraSPARC T1 – наличие в процессоре только одного блока вычислений с плавающей точкой, доступного для всех потоков всех ядер. В процессоре UltraSPARC T2 эту проблему решили – у каждого ядра есть собственный модуль для выполнения вещественных операций. Также в UltraSPARC T2 была поднята максимальная тактовая частота – до 1,4 ГГц. Плюс увеличен объем кэша второго уровня – до 4 Мб, однако в отличие от UltraSPARC T1 кэш не общий, а раздельный – по 512 Кб на каждое ядро. Кроме того, в процессор интегрированы два 10-Gbit контроллера Ethernet и контроллер шины PCI Express.

### 1.3. Ускорители вычислений

Технологический мир сегодня пронизан конвергенцией – взаимным влиянием и даже взаимопроникновением технологий, стиранием границ между ними, возникновением многих интересных результатов на стыке областей в рамках междисциплинарных работ. Одно из проявлений этого явления – «игра» основных производителей аппаратных составляющих компьютеров на «чужих полях». Так компании NVIDIA и ATI (последняя теперь в составе компании AMD), накопив опыт и поняв, что пиковая мощность их продуктов уже стала сравнима с кластерами «средней руки», от выпуска графических ускорителей начали движение на рынок высокопроизводительных решений, представив соответствующие продукты (семейства NVIDIA® Tesla™ и ATI FireStream™). Напротив, компании, традиционно выпускавшие процессоры и серверные решения, взялись осваивать область мультимедиа: компания Intel разрабатывает многопоточные векторные графические устройства, компания IBM в составе альянса STI создала, как мы уже обсуждали, процессор Cell, изначально ориентированный именно на быструю обработку мультимедиа информации. Никуда не делись и типичные ускорители вычислений, способные существенно добавить мощности даже обычным «персоналкам» – на этом направлении работает, например, компания ClearSpeed Technology [108].

Как следствие, перед обычными пользователями, желающими попробовать «с чем едят» суперкомпьютерные технологии возникает большой и не всегда просто осуществляемый выбор. Попытаемся немного прокомментировать возможности двух из представленных выше систем: ClearSpeed™ Advance™ X620 и NVIDIA® Tesla™ D870.

#### 1.3.1. Ускоритель ClearSpeed™ Advance™ X620

ClearSpeed™ Advance™ X620 – это ускоритель операций над данными с плавающей запятой, представленных в формате с двойной точностью.

Ускоритель является сопроцессором, разработанным специально для серверов и рабочих станций, которые основаны на 32-х или 64-х битной x86 архитектуре, и построен на базе двух процессоров CSX600 со 194 вычислительными ядрами. X620 подключается к PCI-X разъему на материнской плате. Среда разработки под X620 основана на языке C и включает SDK, а также набор инструментов для написания и отладки программ.

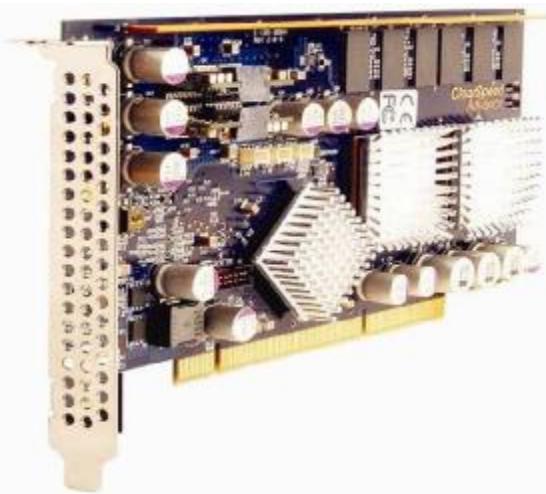


Рис. 1.10. Ускоритель ClearSpeed Advance X620

Технические характеристики X620:

- 2 процессора CSX600, работающих на частоте 210 МГц.
- Каждый CSX600 содержит 97 ядер (из них 96 в poly-исполнительном блоке и 1 в mono-исполнительном блоке).
- Пиковая производительность X620 – 66 гигафлопс.
- Память 1 Гб DDR2 DRAM 64-bit (по 0,5 Гб на каждый процессор).
- Пиковая пропускная способность шины памяти 3,2 Гб/с.
- Подключается к ПК через PCI-X разъем материнской платы.
- Максимальная потребляемая мощность: 43 Вт.
- Рабочий диапазон температур: 10..50°C.

Ускоритель X260 поддерживает операционные системы семейств Linux (Red Hat, SUSE) и Windows (XP, 2003 Server). Для работы с X620 необходимо установить специальный драйвер.

При работе с ускорителем программист может использовать несколько уровней памяти:

- каждый mono-процессор обладает локальной памятью (mono local memory) размером 128 Кб;
- каждый poly-процессор обладает локальной памятью (poly local memory) размером 6 Кб;
- все процессоры во всех блоках имеют доступ к общей памяти устройства (global memory или device memory) размером 1024 Мб.

Главное (хост) приложение, использующее возможности X620 состоит из двух частей:

- программа, выполняющаяся на главном процессоре;
- программа, которая выполняется на ускорителе.

ClearSpeed™ Advance™ X620 обычно используется в качестве сопроцессора для ускорения внутренних циклов программы, достигающегося за счет использования стандартных математических библиотек, разработанных ClearSpeed. Если в приложении происходит вызов функции, поддерживаемой математическими библиотеками ClearSpeed, то библиотека сама анализирует возможность ее ускорения. В зависимости от выполненного анализа функция начинает выполняться либо на главном процессоре, либо перехватывается ускорителем. Этот механизм широко используется в ряде математических приложений, например в MATLAB.

Если в приложении происходит вызов функции, не поддерживаемой библиотеками ClearSpeed, то программист, в случае необходимости, может сам реализовать ее, используя имеющийся инструментарий.

### 1.3.2. Настольный суперкомпьютер NVIDIA® Tesla™ D870

В 2007 году компания NVIDIA представила продукты семейства Tesla™ для построения высокопроизводительных вычислительных систем. Семейство включает вычислительный процессор NVIDIA® Tesla™ C870, приставной суперкомпьютер NVIDIA® Tesla™ D870 и вычислительный сервер NVIDIA® Tesla™ S870. Два последних решения построены на двух и четырех процессорах C870 соответственно. Сервер S870 предназначен для построения на его основе кластерных решений и его обсуждение выходит за рамки данного материала.

Настольный (в терминологии компании NVIDIA - приставной) суперкомпьютер NVIDIA® Tesla™ D870 конструктивно представляет собой два процессора C870,

объединенных в небольшом корпусе. Необходимым условием для подключения D870 к компьютеру является наличие не менее двух PCI Express разъемов, в первом должна быть установлена видеокарта NVIDIA не менее чем 8xxx серии, а во второй ставится специальная плата-переходник, к которой и подключается D870.

Технические характеристики D870:

- 2 платы NVIDIA® Tesla™ C870.
- Каждый процессор C870 содержит 128 скалярных процессора с частотой 1,35 ГГц.
- Пиковая производительность D870 – 1 терафлопс.
- Память 3 Гб GDDR3 384-bit (по 1,5 Гб на каждый процессор).
- Пиковая пропускная способность шины памяти 76,8 Гб/с.
- Подключается кабелем к специальной плате-переходнику, установленной в разъем PCI Express x8 или x16.
- Максимальный уровень шума 40 дБ.
- Максимальная потребляемая мощность 520 Вт.



Рис. 1.11. Внешний вид NVIDIA® Tesla™ D870

Суперкомпьютер D870 поддерживает операционные системы семейств Windows, Linux, Mac OS. В комплект поставки входят следующие компоненты (необходимо устанавливать в указанном порядке): CUDA Driver, CUDA Toolkit, CUDA SDK.

CUDA (Compute Unified Device Architecture) – программно аппаратное решение, позволяющее использовать видеопроцессоры для вычислений общего назначения. Возможность программирования видеопроцессоров компании NVIDIA на CUDA существует, начиная с семейства видеокарт 8xxx. Разработка программ для выполнения на D870 происходит таким же способом.

С точки зрения программиста, D870 представляет собой набор независимых мультипроцессоров. Каждый мультипроцессор состоит из нескольких независимых скалярных процессоров, двух модулей для вычисления математических функций, конвейера, а также общей памяти.

CUDA позволяет создавать специальные функции (ядра, kernels), которые выполняются параллельно различными блоками и потоками, в отличие от обычных С-функций. При запуске ядра блоки распределяются по доступным мультипроцессорам. Мультипроцессор занимается распределением, параллельным выполнением потоков внутри блока и их синхронизацией. Каждый поток независимо исполняется на одном скалярном процессоре с собственным стеком инструкций и памятью.

CUDA предоставляет программисту доступ к нескольким уровням памяти:

- каждый поток обладает локальной памятью;
- все потоки внутри блока имеют доступ к быстрой общей памяти блока, время жизни которой совпадает со временем жизни блока; память блока разбита на страницы, доступ к данным на разных страницах осуществляется параллельно;
- все потоки во всех блоках имеют доступ к общей памяти устройства.

Всем потокам также доступны два вида общей памяти для чтения: константная и текстурная, они кэшируются. Так же как и в общей памяти устройства, данные сохраняются на протяжении работы приложения.

## 1.4. Персональные мини-кластеры

Были времена, когда кластеры собирали на основе обычных рабочих станций. Таков, например, был первый известный кластер Beowulf, собранный летом 1994 года в научно-космическом центре NASA из 16 компьютеров на базе процессоров 486DX4 с тактовой частотой 100 MHz. Связь узлов в этом кластере осуществлялась посредством 10 Мбит/с сети, невероятно медленно по сегодняшним меркам. Однако довольно быстро стало понятно, что каждый отдельный узел кластера не нуждается во всем многообразии комплектующих, из которых состоит обычный «отдельно стоящий» компьютер. В действительности все, что должно быть в каждом узле, – процессор, память, жесткий диск с размером достаточным, чтобы установить на него операционную систему, и сетевой интерфейс. Это понимание привело к тому, что характерным форм-фактором для стоечных серверов, из которых собирают современные кластеры, стал системный блок с высотой порядка 4,5 см. (форм-фактор 1U). Идея «упаковать» некоторое количество узлов в небольших размерах корпуса так, чтобы кластер не требовал отдельного помещения, что называется, лежала на поверхности. Мы уже упоминали про мини-кластер RenderCube. Еще одним примером подобного подхода является мини-кластер, представленный в 2005 году стартап-компанией Orion Multisystems, включающий до 96 процессоров и до 192 Гб памяти [103].

В 2006 году свои решения в этой нише начала поставлять российская компания «Т-Платформы», сначала на основе процессоров компаний AMD, а позднее и компании Intel.



Рис. 1.12. Мини-кластер Orion Multisystems

### 1.4.1. Персональный суперкомпьютер T-Forge Mini

T-Forge Mini – компактный суперкомпьютер, габаритные размеры (360×321×680 мм) и небольшая масса которого позволяют установить его непосредственно на рабочем месте сотрудника. Уровень шума T-Forge Mini не превышает 45 децибел, что позволяет использовать мини-кластер в том числе и в офисных условиях.

Технические характеристики мини-кластера T-Forge Mini:

- До 4-х двухпроцессорных узлов на базе двухъядерных процессоров AMD Opteron™ или AMD Opteron™ HE с низким энергопотреблением, объединенных сетью Gigabit Ethernet.
- Память до 64 Гб.
- До 4-х устройств HDD SATA общим объемом до 2 Тб.
- 4 блока питания мощностью 350 Вт.
- До 9-ти портов Gigabit Ethernet.
- Контроль частоты вращения вентиляторов.
- Видео-карта ATI Rage XL 8Mb.
- Плата удаленного управления сервером с поддержкой стандарта IPMI (опционально).
- Операционная система: ОС SUSE Linux Enterprise Server 9, RedHat Enterprise Linux 4 или Microsoft Windows Compute Cluster Server 2003.



Рис. 1.13. Мини-кластер T-Forge Mini

Один из первых мини-кластеров T-Forge Mini в максимальной конфигурации (не считая объема памяти) с пиковой производительностью в 70 гигафлопс был приобретен ННГУ в рамках нацпроекта «Образование» в 2006 году [105] и используется при выполнении многих образовательных и научных проектов.

#### 1.4.2. Мини-кластер T-Edge Mini

Мини-кластер T-Edge Mini несколько крупнее, чем T-Forge Mini (габаритные размеры T-Edge Mini 530×360×700 мм), да и масса под 100 кг уже не позволяет назвать его «настольным», но под столом он вполне способен разместиться. Дополнительный объем был использован компанией-разработчиком для расширения возможностей кластера - во-первых, размещен дополнительный вычислительный узел, во-вторых, поддерживаются четырехъядерные процессоры, в-третьих, в качестве интерконнекта может быть использован не только Gigabit Ethernet, но и Infiniband.

Технические характеристики мини-кластера T-Edge Mini:

- До 5-ти двухпроцессорных узлов на базе четырехъядерных процессоров Intel® Xeon®, объединенных сетью Gigabit Ethernet или Infiniband.
- Память до 64 Гб.
- 3 блока питания мощностью 600 Вт.
- Адаптер сервисной сети, осуществляющий мониторинг и администрирование управляющего узла по протоколу RS 485 ServNET v.2.0.
- Порты ввода/вывода на передней панели: VGA port, USB ports, Keyboard and Mouse.
- Порты ввода/вывода на задней панели: 2 RJ-45 GbE ports, 1 RJ-45 FE port, 4 порта для мониторинга узлов по протоколу RS 485.
- Встроенный DVD привод.
- Встроенный KVM и GbE коммутатор.
- Операционная система: OC SUSE Linux Enterprise Server 9, RedHat Enterprise Linux 4 или Microsoft Windows Compute Cluster Server 2003.



Рис. 1.14. Мини-кластер T-Forge Mini

В 2007 году ННГУ в рамках нацпроекта «Образование» приобрел два мини-кластера T-Edge Mini, построенных на процессорах Intel® Xeon® 5320 (1,86 ГГц) с 20 Гб оперативной памяти и дисковой подсистемой на 1,25 Тб. В качестве интерконнекта в одном из мини-кластеров использован Gigabit Ethernet, во втором Infiniband. Пиковая производительность каждого – 297 гигафлопс.

### 1.5. Краткий обзор раздела

Раздел посвящен рассмотрению компьютерных вычислительных устройств, построенных на базе многоядерных процессоров для организации высокопроизводительных вычислений.

С этой целью в разделе дается краткая характеристика класса многопроцессорных вычислительных систем MIMD по классификации Флинна. С учетом характера использования оперативной памяти в составе этого класса выделены две важных группы

систем с общей разделяемой и распределенной памятью – *мультипроцессоры* и *мультикомпьютеры*.

Далее в разделе отмечается, что наиболее перспективное направление для достижения высокой производительности вычислений на данный момент времени состоит в явной организации многопроцессорности вычислительных устройств. Для обоснования данного утверждения приводится подробное рассмотрение основных способов организации многопроцессорности – *симметричной мультипроцессорности* (*Symmetric Multiprocessor, SMP*), *одновременной многопотокости* (*Simultaneous Multithreading, SMT*) и *многоядерности* (*multicore*). Сравнение перечисленных подходов позволяет сделать вывод, что многоядерность процессоров становится одной из основных направлений развития компьютерной техники. Значимость такого подхода привела даже к тому, что известный закон Мура теперь формулируется в новом «многоядерном» виде «Количество вычислительных ядер в процессоре будет удваиваться каждые 18 месяцев».

В продолжение темы многоядерности далее в подразделе 1.2 дается характеристика ряда конкретных широко применяемых в данный момент времени многоядерных процессоров основных компаний-разработчиков Intel, AMD, IBM, Sun.

Для полноты картины в подразделе 1.3 приводится описание ряда аппаратных устройств (видеокарт и вычислительных сопроцессоров), которые могут быть использованы для существенного ускорения вычислений. И для завершения рассматриваемой темы в подразделе 1.4 дается краткая характеристика «персональных» мини-кластеров, которые позволяют при достаточно «экономных» финансовых затратах приступить к решению имеющихся вычислительно-трудоемких задач с использованием высокопроизводительных вычислительных систем.

## 1.6. Обзор литературы

Дополнительная информация об архитектуре параллельных вычислительных систем может быть получена, например, [8, 18, 49, 67, 78]; полезная информация содержится также в [47, 100].

Информация по новейшим разработкам многоядерных процессоров содержится на официальных сайтах компаний-производителей компьютерного оборудования Intel, AMD, IBM, Sun, Nvidia, ClearSpeed и др.

Подробное рассмотрение вопросов, связанных с построением и использованием кластерных вычислительных систем, проводится в [47, 100]. Практические рекомендации по построению кластеров для разных систем платформ могут быть найдены в [19, 92-93].

## 1.7. Контрольные вопросы

1. В чем заключаются основные способы достижения параллелизма?
2. В чем могут состоять различия параллельных вычислительных систем?
3. Что положено в основу классификация Флинна?
4. В чем состоит принцип разделения многопроцессорных систем на мультипроцессоры и мультикомпьютеры?
5. Какие классы систем известны для мультипроцессоров?
6. В чем состоят положительные и отрицательные стороны симметричных мультипроцессоров?
7. Чем обосновывается целесообразность аппаратной поддержки одновременной многопотоковости (simultaneous multithreading)?
8. Какое ускорение вычислений может достигаться для процессоров с поддержкой одновременной многопотоковости?

9. Чем вызывается необходимость разработки многоядерных процессоров? Приведите положительные и отрицательные стороны многоядерности.
10. Какие требования должны быть предъявлены к программам для их эффективного выполнения на многоядерных процессорах?
11. В чем могут состоять различия конкретных многоядерных процессоров? Приведите несколько примеров.
12. Чем вызвана необходимость разработки ускорителей вычислений общего назначения?
13. Каким образом графические процессоры могут быть использованы для организации высокопроизводительных вычислений?
14. Какие характерные признаки отличают персональные мини-кластеры?

## **1.8. Задачи и упражнения**

1. Приведите дополнительные примеры параллельных вычислительных систем.
2. Выполните рассмотрение дополнительных способов классификации компьютерных систем.
3. Рассмотрите способы обеспечения когерентности кэшей в системах с общей разделяемой памятью.
4. Приведите дополнительные примеры многоядерных процессоров.
5. Изучите и дайте общую характеристику способов разработки программ для графических процессоров (на примере настольного суперкомпьютера NVIDIA Tesla).
6. Изучите и дайте общую характеристику способов разработки программ для ускорителей вычислений общего назначения (на примере вычислительных сопроцессоров компании ClearSpeed)?

Глава 2. Моделирование и анализ параллельных вычислений .....	1
2.1. Модель вычислений в виде графа "операции-операнды" .....	1
2.2. Описание схемы параллельного выполнения алгоритма.....	2
2.3. Определение времени выполнения параллельного алгоритма .....	3
2.4. Показатели эффективности параллельного алгоритма .....	6
2.5. Вычисление частных сумм последовательности числовых значений.....	7
2.5.1. Последовательный алгоритм суммирования .....	7
2.5.2. Каскадная схема суммирования .....	8
2.5.3. Модифицированная каскадная схема .....	9
2.5.4. Вычисление всех частных сумм .....	10
2.6. Оценка максимально достижимого параллелизма .....	12
2.7. Анализ масштабируемости параллельных вычислений .....	14
2.8. Краткий обзор раздела .....	15
2.9. Обзор литературы .....	16
2.10. Контрольные вопросы .....	16
2.11. Задачи и упражнения.....	17

## **Глава 2.**

### **Моделирование и анализ параллельных вычислений**

При разработке параллельных алгоритмов решения сложных научно-технических задач принципиальным моментом является анализ эффективности использования параллелизма, состоящий обычно в оценке получаемого ускорения процесса вычислений (сокращения времени решения задачи). Формирование подобных оценок ускорения может осуществляться применительно к выбранному вычислительному алгоритму (оценка эффективности распараллеливания конкретного алгоритма). Другой важный подход может состоять в построении оценок максимально возможного ускорения процесса решения задачи конкретного типа (оценка эффективности параллельного способа решения задачи).

В данном разделе описывается модель вычислений в виде графа "операции-операнды", которая может использоваться для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач, приводятся оценки эффективности максимально возможного параллелизма, которые могут быть получены в результате анализа имеющихся моделей вычислений. Примеры использования излагаемого теоретического материала приводятся при рассмотрении параллельных алгоритмов в завершающих разделах настоящего учебного материала.

#### **2.1. Модель вычислений в виде графа "операции-операнды"**

Для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач может быть использована модель в виде графа "операции-операнды" (см., например, [8,10,44]). Для уменьшения сложности излагаемого материала при построении модели будет предполагаться, что время выполнения любых вычислительных операций является одинаковым и равняется 1 (в тех или иных единицах измерения). Кроме того, принимается, что передача данных между вычислительными устройствами выполняется мгновенно без каких-либо затрат времени (что может быть справедливо, например, при наличии общей разделяемой памяти в параллельной вычислительной системе).

Представим множество операций, выполняемых в исследуемом алгоритме решения вычислительной задачи, и существующие между операциями информационные зависимости в виде ациклического ориентированного графа

$$G = (V, R),$$

где  $V = \{1, \dots, |V|\}$  есть множество вершин графа, представляющих выполняемые операции алгоритма, а  $R$  есть множество дуг графа (при этом дуга  $r = (i, j)$  принадлежит графу только, если операция  $j$  использует результат выполнения операции  $i$ ). Для примера на рис. 2.1 показан граф алгоритма вычисления площади прямоугольника, заданного координатами двух противолежащих углов. Как можно заметить по приведенному примеру, для выполнения выбранного алгоритма решения задачи могут быть использованы разные схемы вычислений и построены соответственно разные вычислительные модели. Как будет показано далее, разные схемы вычислений обладают различными возможностями для распараллеливания и, тем самым, при построении модели вычислений может быть поставлена задача выбора наиболее подходящей для параллельного исполнения вычислительной схемы алгоритма.

В рассматриваемой вычислительной модели алгоритма вершины без входных дуг могут использоваться для задания операций ввода, а вершины без выходных дуг – для операций вывода. Обозначим через  $\bar{V}$  множество вершин графа без вершин ввода, а через  $d(G)$  диаметр (длину максимального пути) графа.

## 2.2. Описание схемы параллельного выполнения алгоритма

Операции алгоритма, между которыми нет пути в рамках выбранной схемы вычислений, могут быть выполнены параллельно (для вычислительной схемы на рис. 2.1, например, параллельно могут быть реализованы сначала все операции умножения, а затем первые две операции вычитания). Возможный способ описания параллельного выполнения алгоритма может состоять в следующем (см., например, [8,10,44]).

Пусть  $p$  есть количество процессоров, используемых для выполнения алгоритма. Тогда для параллельного выполнения вычислений необходимо задать множество (*расписание*)

$$H_p = \{(i, P_i, t_i) : i \in V\},$$

$$\begin{array}{c}
 (x_2, y_2) \\
 \boxed{\phantom{...}} \\
 (x_1, y_1)
 \end{array}
 \quad S = ((x_2 - x_1)(y_2 - y_1)) = \\
 = x_2 y_2 - x_2 y_1 - x_1 y_2 + x_1 y_1$$

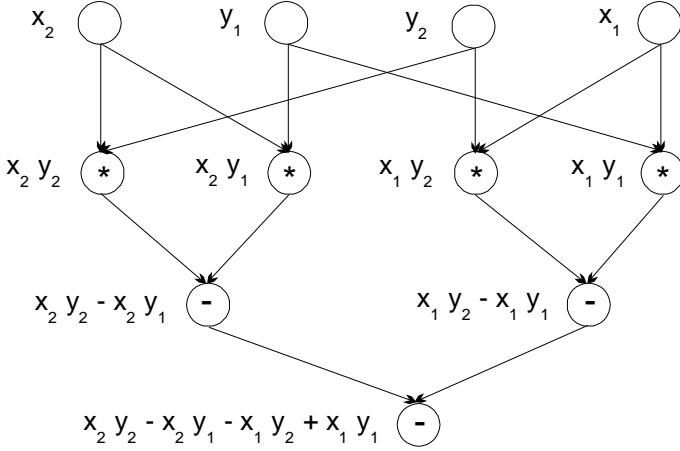


Рис. 2.1. Пример вычислительной модели алгоритма в виде графа "операции-операнды"

в котором для каждой операции  $i \in V$  указывается номер используемого для выполнения операции процессора  $P_i$  и время начала выполнения операции  $t_i$ . Для того, чтобы расписание было реализуемым, необходимо выполнение следующих требований при задании множества  $H_p$ :

- 1)  $\forall i, j \in V : t_i = t_j \Rightarrow P_i \neq P_j$ , т.е. один и тот же процессор не должен назначаться разным операциям в один и тот же момент времени,
- 2)  $\forall (i, j) \in R \Rightarrow t_j \geq t_i + 1$ , т.е. к назначаемому моменту выполнения операции все необходимые данные уже должны быть вычислены.

### 2.3. Определение времени выполнения параллельного алгоритма

Вычислительная схема алгоритма  $G$  совместно с расписанием  $H_p$  может рассматриваться как модель параллельного алгоритма  $A_p(G, H_p)$ , исполняемого с использованием  $p$  процессоров. Время выполнения параллельного алгоритма определяется максимальным значением времени, используемым в расписании

$$T_p(G, H_p) = \max_{i \in V} (t_i + 1).$$

Для выбранной схемы вычислений желательно использование расписания, обеспечивающего минимальное время исполнения алгоритма

$$T_p(G) = \min_{H_p} T_p(G, H_p).$$

Уменьшение времени выполнения может быть обеспечено и путем подбора наилучшей вычислительной схемы

$$T_p = \min_G T_p(G).$$

Оценки  $T_p(G, H_p)$ ,  $T_p(G)$  и  $T_p$  могут быть использованы в качестве показателей времени выполнения параллельного алгоритма. Кроме того, для анализа максимально возможного параллелизма можно определить оценку наиболее быстрого исполнения алгоритма

$$T_\infty = \min_{p \geq 1} T_p.$$

Оценку  $T_\infty$  можно рассматривать как минимально возможное время выполнения параллельного алгоритма при использовании неограниченного количества процессоров (концепция вычислительной системы с бесконечным количеством процессоров, обычно называемой *паракомпьютером*, широко используется при теоретическом анализе параллельных вычислений).

Оценка  $T_1$  определяет время выполнения алгоритма при использовании одного процессора и представляет, тем самым, время выполнения последовательного варианта алгоритма решения задачи. Построение подобной оценки является важной задачей при анализе параллельных алгоритмов, поскольку она необходима для определения эффекта использования параллелизма (ускорения времени решения задачи). Очевидно, что

$$T_1(G) = |\bar{V}|,$$

где  $|\bar{V}|$ , напомним, есть количество вершин вычислительной схемы  $G$  без вершин ввода. Важно отметить, что если при определении оценки  $T_1$  ограничиться рассмотрением только одного выбранного алгоритма решения задачи и использовать величину

$$T_1 = \min_G T_1(G),$$

то получаемые при использовании такой оценки показатели ускорения будут характеризовать эффективность распараллеливания выбранного алгоритма. Для оценки эффективности параллельного решения исследуемой задачи вычислительной математики время последовательного решения следует определять с учетом различных последовательных алгоритмов, т.е. использовать величину

$$T_1^* = \min T_1,$$

где операция минимума берется по множеству всех возможных последовательных алгоритмов решения данной задачи.

Приведем без доказательства теоретические положения, характеризующие свойства оценок времени выполнения параллельного алгоритма (см. [44]).

**Теорема 1.** Минимально возможное время выполнения параллельного алгоритма определяется длиной максимального пути вычислительной схемы алгоритма, т.е.

$$T_\infty(G) = d(G).$$

**Теорема 2.** Пусть для некоторой вершины вывода в вычислительной схеме алгоритма существует путь из каждой вершины ввода. Кроме того, пусть входная степень вершин схемы (количество входящих дуг) не превышает 2. Тогда минимально возможное время выполнения параллельного алгоритма ограничено снизу значением

$$T_\infty(G) = \log_2 n,$$

где  $n$  есть количество вершин ввода в схеме алгоритма.

**Теорема 3.** При уменьшении числа используемых процессоров время выполнения алгоритма увеличивается пропорционально величине уменьшения количества процессоров, т.е.

$$\forall q = cp, \quad 0 < c < 1 \Rightarrow T_p \leq cT_q.$$

**Теорема 4.** Для любого количества используемых процессоров справедлива следующая верхняя оценка для времени выполнения параллельного алгоритма

$$\forall p \Rightarrow T_p < T_\infty + T_1 / p.$$

**Теорема 5.** Времени выполнения алгоритма, которое сопоставимо с минимально возможным временем  $T_\infty$ , можно достичь при количестве процессоров порядка  $p \sim T_1 / T_\infty$ , а именно,

$$p \geq T_1 / T_\infty \Rightarrow T_p \leq 2T_\infty.$$

При меньшем количестве процессоров время выполнения алгоритма не может превышать более, чем в 2 раза, наилучшее время вычислений при имеющемся числе процессоров, т.е.

$$p < T_1 / T_\infty \Rightarrow \frac{T_1}{p} \leq T_p \leq 2 \frac{T_1}{p}.$$

Приведенные утверждения позволяют дать следующие рекомендации по правилам формирования параллельных алгоритмов:

- 1) при выборе вычислительной схемы алгоритма должен использоваться граф с минимально возможным диаметром (см. теорему 1);
- 2) для параллельного выполнения целесообразное количество процессоров определяется величиной  $p \sim T_1 / T_\infty$  (см. теорему 5);
- 3) время выполнения параллельного алгоритма ограничивается сверху величинами, приведенными в теоремах 4 и 5.

Для вывода рекомендаций по формированию расписания по параллельному выполнению алгоритма приведем доказательство теоремы 4.

**Доказательство теоремы 4.** Пусть  $H_\infty$  есть расписание для достижения минимально возможного времени выполнения  $T_\infty$ . Для каждой итерации  $t$ ,  $0 \leq t \leq T_\infty$ , выполнения расписания  $H_\infty$  обозначим через  $n_t$  количество операций, выполняемых в ходе итерации  $t$ . Расписание выполнения алгоритма с использованием  $p$  процессоров может быть построено следующим образом. Выполнение алгоритма разделим на  $T_\infty$  шагов; на каждом шаге  $t$  следует выполнить все  $n_t$  операций, которые выполнялись на итерации  $t$  расписания  $H_\infty$ . Выполнение этих операций может быть выполнено не более, чем за  $\lceil n_t / p \rceil$  итераций при использовании  $p$  процессоров. Как результат, время выполнения алгоритма  $T_p$  может быть оценено следующим образом

$$T_p = \sum_{t=1}^{T_\infty} \left\lceil \frac{n_t}{p} \right\rceil < \sum_{t=1}^{T_\infty} \left( \frac{n_t}{p} + 1 \right) = \frac{T_1}{p} + T_\infty.$$

Доказательство теоремы дает практический способ построения расписания параллельного алгоритма. Первоначально может быть построено расписание без учета ограниченности числа используемых процессоров (расписание для паракомпьютера). Затем, согласно схеме вывода теоремы, может быть построено расписание для конкретного количества процессоров.

## 2.4. Показатели эффективности параллельного алгоритма

**Ускорение (speedup)**, получаемое при использовании параллельного алгоритма для  $p$  процессоров, по сравнению с последовательным вариантом выполнения вычислений определяется величиной

$$S_p(n) = T_1(n)/T_p(n),$$

т.е. как отношение времени решения задач на скалярной ЭВМ к времени выполнения параллельного алгоритма (величина  $n$  используется для параметризации вычислительной сложности решаемой задачи и может пониматься, например, как количество входных данных задачи).

**Эффективность (efficiency)** использования параллельным алгоритмом процессоров при решении задачи определяется соотношением

$$E_p(n) = T_1(n)/(pT_p(n)) = S_p(n)/p$$

(величина эффективности определяет среднюю долю времени выполнения алгоритма, в течение которой процессоры реально используются для решения задачи).

Из приведенных соотношений можно показать, что в наилучшем случае  $S_p(n) = p$  и  $E_p(n) = 1$ . При практическом применении данных показателей для оценки эффективности параллельных вычислений следует учитывать следующих два важных момента:

- При определенных обстоятельствах ускорение может оказаться больше числа используемых процессоров  $S_p(n) > p$  - в этом случае говорят о существовании *сверхлинейного (superlinear)* ускорения. Несмотря на парадоксальность таких ситуаций (ускорение превышает число процессоров), на практике сверхлинейное ускорение может иметь место. Одной из причин такого явления может быть неравноправность выполнения последовательной и параллельной программ. Например, при решении задачи на одном процессоре оказывается недостаточно оперативной памяти для хранения всех обрабатываемых данных и, как результат, необходимым становится использование более медленной внешней памяти (в случае же использования нескольких процессоров оперативной памяти может оказаться достаточно за счет разделения данных между процессорами). Еще одной причиной сверхлинейного ускорения может быть нелинейный характер зависимости сложности решения задачи в зависимости от объема обрабатываемых данных. Так, например, известный алгоритм пузырьковой сортировки характеризуется квадратичной зависимостью количества необходимых операций от числа упорядочиваемых данных. Как результат, при распределении сортируемого массива между процессорами может быть получено ускорение, превышающее число процессоров (более подробно данный пример рассматривается в разделе 9). Источником сверхлинейного ускорения может быть и различие вычислительных схем последовательного и параллельного методов.

- При внимательном рассмотрении можно обратить внимание, что попытки повышения качества параллельных вычислений по одному из показателей (ускорению или эффективности) может привести к ухудшению ситуации по другому показателю,

ибо показатели качества параллельных вычислений являются противоречивыми. Так, например, повышение ускорения обычно может быть обеспечено за счет увеличения числа процессоров, что приводит, как правило, к падению эффективности. И, обратно, повышение эффективности достигается во многих случаях при уменьшении числа процессоров (в предельном случае идеальная эффективность  $E_p(n)=1$  легко обеспечивается при использовании одного процессора). Как результат, разработка методов параллельных вычислений часто предполагает выбор некоторого компромиссного варианта с учетом желаемых показателей ускорения и эффективности.

При выборе надлежащего параллельного способа решения задачи может оказаться полезной оценка **стоимости** (*cost*) вычислений, определяемой как произведение времени параллельного решения задачи и числа используемых процессоров

$$C_p = pT_p .$$

В этой связи можно определить понятие **стоимостно-оптимального** (*cost-optimal*) параллельного алгоритма как метода, стоимость которого является пропорциональной времени выполнения наилучшего последовательного алгоритма.

Далее для иллюстрации введенных понятий в следующем пункте будет рассмотрен учебный пример решения задачи вычисления частных сумм для последовательности числовых значений. Кроме того, данные показатели будут использоваться для характеристики эффективности всех рассматриваемых далее параллельных алгоритмов при решении типовых задач вычислительной математики.

## 2.5. Вычисление частных сумм последовательности числовых значений

Рассмотрим для демонстрации ряда проблем, возникающих при разработке параллельных методов вычислений, сравнительно простую задачу *нахождения частных сумм* последовательности числовых значений

$$S_k = \sum_{i=1}^k x_i, \quad 1 \leq k \leq n ,$$

где  $n$  есть количество суммируемых значений (данная задача известна также под названием *prefix sum problem*).

Изучение возможных параллельных методов решения данной задачи начнем с еще более простого варианта ее постановки – с задачи *вычисления общей суммы* имеющегося набора значений (в таком виде задача суммирования является частным случаем общей задачи *редукции*)

$$S = \sum_{i=1}^n x_i .$$

### 2.5.1. Последовательный алгоритм суммирования

Традиционный алгоритм для решения этой задачи состоит в последовательном суммировании элементов числового набора

$$S = 0,$$

$$S = S + x_1, \dots$$

Вычислительная схема данного алгоритма может быть представлена следующим образом (см. рис. 2.2):

$$G_1 = (V_1, R_1),$$

где  $V_1 = \{v_{01}, \dots, v_{0n}, v_{11}, \dots, v_{1n}\}$  есть множество операций (вершины  $v_{01}, \dots, v_{0n}$  обозначают операции ввода, каждая вершина  $v_{li}$ ,  $1 \leq i \leq n$ , соответствует прибавлению значения  $x_i$  к накапливающейся сумме  $S$ ), а

$$R_1 = \{(v_{0i}, v_{1i}), (v_{1i}, v_{1i+1}), \quad 1 \leq i \leq n-1\}$$

есть множество дуг, определяющих информационные зависимости операций.

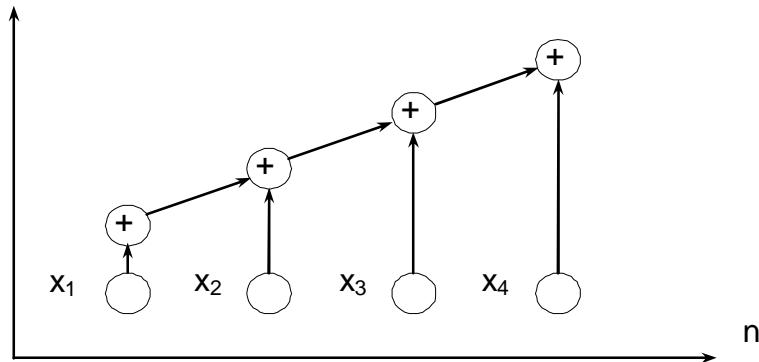


Рис. 2.2. Последовательная вычислительная схема алгоритма суммирования

Как можно заметить, данний "стандартный" алгоритм суммирования допускает только строго последовательное исполнение и не может быть распараллелен.

### 2.5.2. Каскадная схема суммирования

Параллелизм алгоритма суммирования становится возможным только при ином способе построения процесса вычислений, основанном на использовании ассоциативности операции сложения. Получаемый новый вариант суммирования (известный в литературе как *каскадная схема*) состоит в следующем (см. рис. 2.3):

- на первой итерации каскадной схемы все исходные данные разбиваются на пары, и для каждой пары вычисляется сумма их значений,
- далее все полученные суммы также разбиваются на пары, и снова выполняется суммирование значений пар и т.д.

Данная вычислительная схема может быть определена как граф (пусть  $n = 2^k$ )

$$G_2 = (V_2, R_2),$$

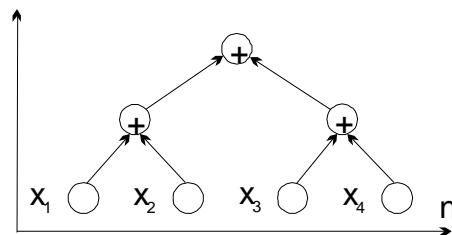


Рис. 2.3. Каскадная схема алгоритма суммирования

где  $V_2 = \{(v_{i1}, \dots, v_{il_i}), \quad 0 \leq i \leq k, \quad 1 \leq l_i \leq 2^{-i} n\}$  есть вершины графа ( $(v_{01}, \dots, v_{0n})$  - операции ввода,  $(v_{11}, \dots, v_{1n/2})$  - операции первой итерации и т.д.), а множество дуг графа определяется соотношениями:

$$R_2 = \{(v_{i-1,2^{j-1}} v_{ij}), (v_{i-1,2^j} v_{ij}), \quad 1 \leq i \leq k, \quad 1 \leq j \leq 2^{-i} n\}.$$

Как нетрудно оценить, количество итераций каскадной схемы оказывается равным величине

$$k = \log_2 n,$$

а общее количество операций суммирования

$$K_{nosl} = n/2 + n/4 + \dots + 1 = n - 1$$

совпадает с количеством операций последовательного варианта алгоритма суммирования. При параллельном исполнении отдельных итераций каскадной схемы общее количество параллельных операций суммирования является равным

$$K_{nap} = \log_2 n.$$

Поскольку считается, что время выполнения любых вычислительных операций является одинаковым и единичным, то  $T_1 = K_{nosl}$ ,  $T_p = K_{nap}$ , поэтому показатели ускорения и эффективности каскадной схемы алгоритма суммирования можно оценить как

$$S_p = T_1 / T_p = (n - 1) / \log_2 n,$$

$$E_p = T_1 / pT_p = (n - 1) / (p \log_2 n) = (n - 1) / ((n/2) \log_2 n),$$

где  $p = n/2$  есть необходимое для выполнения каскадной схемы количество процессоров.

Анализируя полученные характеристики, можно отметить, что время параллельного выполнения каскадной схемы совпадает с оценкой для параллельного компьютера в теореме 2. Однако при этом эффективность использования процессоров уменьшается при увеличении количества суммируемых значений

$$\lim E_p \rightarrow 0 \quad \text{при} \quad n \rightarrow \infty.$$

### 2.5.3. Модифицированная каскадная схема

Получение асимптотически ненулевой эффективности может быть обеспечено, например, при использовании модифицированной каскадной схемы (см. [44]). Для упрощения построения оценок можно предположить  $n = 2^k$ ,  $k = 2^s$ . Тогда в новом варианте каскадной схемы все проводимые вычисления подразделяются на два последовательно выполняемых этапа суммирования (см. рис. 2.4):

- на первом этапе вычислений все суммируемые значения подразделяются на  $(n/\log_2 n)$  групп, в каждой из которых содержится  $\log_2 n$  элементов; далее для каждой группы вычисляется сумма значений при помощи последовательного алгоритма суммирования; вычисления в каждой группе могут выполняться независимо друг от друга (т.е. параллельно – для этого необходимо наличие не менее  $(n/\log_2 n)$  процессоров);
- на втором этапе для полученных  $(n/\log_2 n)$  сумм отдельных групп применяется обычная каскадная схема.

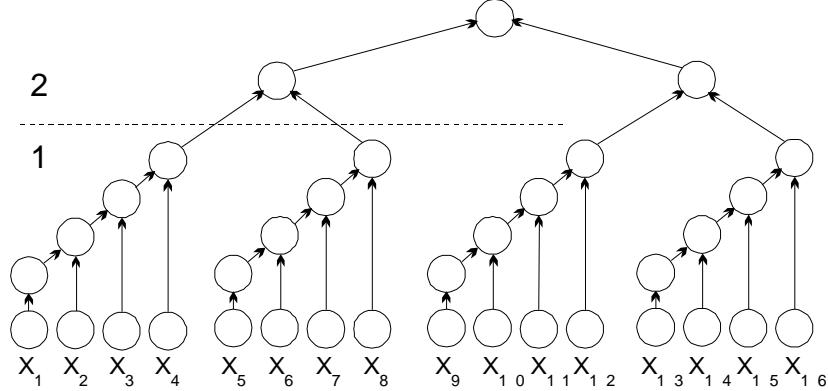


Рис. 2.4. Модифицированная каскадная схема суммирования

Тогда для выполнения первого этапа требуется выполнение  $\log_2 n$  параллельных операций при использовании  $p_1 = (n / \log_2 n)$  процессоров. Для выполнения второго этапа необходимо

$$\log_2(n / \log_2 n) \leq \log_2 n$$

параллельных операций для  $p_2 = (n / \log_2 n) / 2$  процессоров. Как результат, данный способ суммирования характеризуется следующими показателями:

$$T_p = 2 \log_2 n, \quad p = (n / \log_2 n).$$

С учетом полученных оценок показатели ускорения и эффективности модифицированной каскадной схемы определяются соотношениями:

$$S_p = T_1 / T_p = (n - 1) / 2 \log_2 n,$$

$$E_p = T_1 / p T_p = (n - 1) / (2(n / \log_2 n) \log_2 n) = (n - 1) / 2n.$$

Сравнивая данные оценки с показателями обычной каскадной схемы, можно отметить, что ускорение для предложенного параллельного алгоритма уменьшилось в 2 раза, однако для эффективности нового метода суммирования можно получить асимптотически ненулевую оценку снизу

$$E_p = (n - 1) / 2n \geq 0.25, \quad \lim E_p \rightarrow 0.5 \quad \text{при} \quad n \rightarrow \infty.$$

Можно отметить также, что данные значения показателей достигаются при количестве процессоров, определенном в теореме 5. Кроме того, необходимо подчеркнуть, что, в отличие от обычной каскадной схемы, модифицированный каскадный алгоритм является стоимостно-оптимальным, поскольку стоимость вычислений в этом случае

$$C_p = p T_p = (n / \log_2 n)(2 \log_2 n)$$

является пропорциональной времени выполнения последовательного алгоритма.

#### 2.5.4. Вычисление всех частных сумм

Вернемся к исходной задаче вычисления всех частных сумм последовательности значений и проведем анализ возможных способов последовательной и параллельной организации вычислений. Вычисление всех частных сумм на скалярном компьютере

может быть получено при помощи обычного последовательного алгоритма суммирования при том же количестве операций (!)

$$T_1 = n.$$

При параллельном исполнении применение каскадной схемы в явном виде не приводит к желаемым результатам; достижение эффективного распараллеливания требует привлечения новых подходов (может быть, даже не имеющих аналогов при последовательном программировании) для разработки новых параллельно-ориентированных алгоритмов решения задач. Так, для рассматриваемой задачи нахождения всех частных сумм, алгоритм, обеспечивающий получение результатов за  $\log_2 n$  параллельных операций (как и в случае вычисления общей суммы), может состоять в следующем (см. рис. 2.5) [44]:

- перед началом вычислений создается копия  $S$  вектора суммируемых значений ( $S = x$ );
- далее на каждой итерации суммирования  $i$ ,  $1 \leq i \leq \log_2 n$ , формируется вспомогательный вектор  $Q$  путем сдвига вправо вектора  $S$  на  $2^{i-1}$  позиций (освобождающиеся при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования векторов  $S$  и  $Q$ .

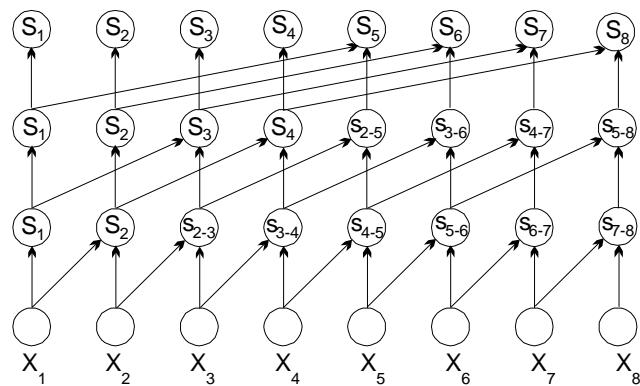


Рис. 2.5. Схема параллельного алгоритма вычисления всех частных сумм (величины  $S_{i-j}$  означают суммы значений от  $i$  до  $j$  элементов числовой последовательности)

Всего параллельный алгоритм выполняется за  $\log_2 n$  параллельных операций сложения. На каждой итерации алгоритма параллельно выполняются  $n$  скалярных операций сложения и, таким образом, общее количество выполняемых скалярных операций определяется величиной

$$K_{nap} = n \log_2 n$$

(параллельный алгоритм содержит большее (!) количество операций по сравнению с последовательным способом суммирования). Необходимое количество процессоров определяется количеством суммируемых значений ( $p = n$ ).

С учетом полученных соотношений, показатели ускорения и эффективности параллельного алгоритма вычисления всех частных сумм оцениваются следующим образом

$$S_p = T_1 / T_p = n / \log_2 n,$$

$$E_p = T_1 / pT_p = n / (p \log_2 n) = n / (n \log_2 n) = 1 / \log_2 n.$$

Как следует из построенных оценок, эффективность алгоритма также уменьшается при увеличении числа суммируемых значений и при необходимости повышения величины этого показателя может оказаться полезной модификация алгоритма, как и в случае с обычной каскадной схемой.

## 2.6. Оценка максимально достижимого параллелизма

Оценка качества параллельных вычислений предполагает знание *наилучших (максимально достижимых)* значений показателей ускорения и эффективности, однако, получение идеальных величин  $S_p = p$  для ускорения и  $E_p = 1$  для эффективности может быть обеспечено не для всех вычислительно трудоемких задач. Так, для рассматриваемого учебного примера в предыдущем пункте минимально достижимое время параллельного вычисления суммы числовых значений составляет  $\log_2 n$ . Определенное содействие в решении данной проблемы могут оказать теоретические утверждения, приведенные в начале данного раздела. В дополнение к ним рассмотрим еще ряд закономерностей, которые могут быть чрезвычайно полезны при построении оценок максимально достижимого параллелизма.

**1. Закон Амдаля.** Достижению максимального ускорения может препятствовать существование в выполняемых вычислениях последовательных расчетов, которые не могут быть распараллелены. Пусть  $f$  есть *доля последовательных вычислений* в применяемом алгоритме обработки данных, тогда в соответствии с *законом Амдаля (Amdahl)* ускорение процесса вычислений при использовании  $p$  процессоров ограничивается величиной

$$S_p \leq \frac{1}{f + (1-f)/p} \leq S^* = \frac{1}{f}.$$

Так, например, при наличии всего 10% последовательных команд в выполняемых вычислениях, эффект использования параллелизма не может превышать 10-кратного ускорения обработки данных. Для рассмотренного учебного примера вычисления суммы значений для каскадной схемы доля последовательных расчетов составляет  $f = \log_2 n / n$  и, как результат, величина возможного ускорения ограничена оценкой  $S^* = n / \log_2 n$ .

Закон Амдаля характеризует одну из самых серьезных проблем в области параллельного программирования (алгоритмов без определенной доли последовательных команд практически не существует). Однако часто доля последовательных действий характеризует не возможность параллельного решения задач, а последовательные свойства применяемых алгоритмов. Как результат, доля последовательных вычислений может быть существенно снижена при выборе более подходящих для распараллеливания методов.

Следует отметить также, что рассмотрение закона Амдаля происходит при предположении, что доля последовательных расчетов  $f$  является постоянной величиной и не зависит от параметра  $n$ , определяющего вычислительную сложность решаемой задачи. Однако для большого ряда задач доля  $f=f(n)$  является убывающей функцией от  $n$ , и в этом случае ускорение для фиксированного числа процессоров может быть увеличено за счет увеличения вычислительной сложности решаемой задачи. Данное замечание может быть сформулировано как утверждение, что ускорение  $S_p = S_p(n)$

является возрастающей функцией от параметра  $n$  (данное утверждение часто именуется как *эффект Амдаля*). Так, например, для учебного примера вычисления суммы значений при использовании фиксированного числа процессоров  $p$  суммируемый набор данных может быть разделен на блоки размера  $n/p$ , для которых сначала параллельно могут быть вычислены частные суммы, а далее эти суммы можно сложить при помощи каскадной схемы. Длительность последовательной части выполняемых операций (минимально-возможное время параллельного исполнения) в этом случае составляет

$$T_p = (n/p) + \log_2 p,$$

что приводит к оценке доли последовательных расчетов как величины

$$f = (1/p) + \log_2 p/n.$$

Как следует из полученного выражения, доля последовательных расчетов  $f$  убывает с ростом  $n$  и в предельном случае мы получаем идеальную оценку максимально возможного ускорения  $S^* = p$ .

**2. Закон Густавсона - Барсиса.** Оценим максимально достижимое ускорение исходя из имеющейся доли последовательных расчетов в выполняемых параллельных вычислениях:

$$g = \frac{t(n)}{t(n) + p(n)/p},$$

где  $t(n)$  и  $p(n)$  есть времена последовательной и параллельной частей выполняемых вычислений соответственно, т.е.

$$T_1 = t(n) + p(n), \quad T_p = t(n) + p(n)/p.$$

С учетом введенной величины  $g$  можно получить

$$t(n) = g \times t(n) + p(n)/p, \quad p(n) = (1-g)p \times t(n) + p(n)/p,$$

что позволяет построить оценку для ускорения

$$S_p = \frac{T_1}{T_p} = \frac{t(n) + p(n)}{t(n) + p(n)/p} = \frac{(t(n) + p(n)/p)(g + (1-g)p)}{t(n) + p(n)/p},$$

которая после упрощения приводится к виду *закона Густавсона-Барсиса (Gustafson-Barsis's law)* [85]

$$S_p = g + (1-g)p = p + (1-p)g.$$

Применительно к учебному примеру суммирования значений при использовании  $p$  процессоров время параллельного выполнения, как уже отмечалось выше, составляет

$$T_p = (n/p) + \log_2 p,$$

что соответствует последовательной доле

$$g = \frac{\log_2 p}{(n/p) + \log_2 p}.$$

За счет увеличения числа суммируемых значений величина  $g$  может быть пренебрежимо малой, обеспечивая получение идеального возможного ускорения  $S_p = p$ .

При рассмотрении закона Густавсона-Барсиса следует учитывать еще один важный момент. При увеличении числа используемых процессоров темп уменьшения времени параллельного решения задач может падать (после превышения определенного порога). Однако при этом за счет уменьшения времени вычислений сложность решаемых задач может быть увеличена (так, например, для учебной задачи суммирования может быть увеличен размер складываемого набора значений). Оценку получаемого при этом ускорения можно определить при помощи сформулированных закономерностей. Такая аналитическая оценка тем более полезна, поскольку решение таких более сложных вариантов задач на одном процессоре может оказаться достаточно трудоемким и даже невозможным, например, в силу нехватки оперативной памяти. С учетом указанных обстоятельств оценку ускорения, получаемую в соответствии с законом Густавсона-Барсиса, еще называют *ускорением масштабирования* (*scaled speedup*), поскольку данная характеристика может показать, насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач.

## 2.7. Анализ масштабируемости параллельных вычислений

Целью применения параллельных вычислений во многих случаях является не только уменьшение времени выполнения расчетов, но и обеспечение возможности решения более сложных вариантов решаемых задач (таких постановок, решение которых не представляется возможным при использовании однопроцессорных вычислительных систем). Способность параллельного алгоритма эффективно использовать процессоры при повышении сложности вычислений является важной характеристикой выполняемых расчетов. В связи с этим, параллельный алгоритм называют *масштабируемым* (*scalable*), если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении постоянного уровня эффективности использования процессоров. Возможный способ характеристики свойств масштабируемости состоит в следующем.

Оценим *накладные расходы* (*total overhead*), которые имеют место при выполнении параллельного алгоритма

$$T_0 = pT_p - T_1.$$

Накладные расходы появляются за счет необходимости организации взаимодействия процессоров, выполнения некоторых дополнительных действий, синхронизации параллельных вычислений и т.п. Используя введенное обозначение, можно получить новые выражения для времени параллельного решения задачи и соответствующего ускорения:

$$T_p = \frac{T_1 + T_0}{p}, \quad S_p = \frac{T_1}{T_p} = \frac{pT_1}{T_1 + T_0}.$$

Используя полученные соотношения, эффективность использования процессоров можно выразить как

$$E_p = \frac{S_p}{p} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0/T_1}.$$

Последнее выражение показывает, что, если сложность решаемой задачи является фиксированной ( $T_1 = const$ ), то при росте числа процессоров эффективность, как правило, будет убывать за счет роста накладных расходов  $T_0$ . При фиксации числа процессоров эффективность использования процессоров можно улучшить путем повышения сложности решаемой задачи  $T_1$  (предполагается, что при росте параметра

сложности  $n$  накладные расходы  $T_0$  увеличиваются медленнее, чем объем вычислений  $T_1$ ). Как результат, при увеличении числа процессоров в большинстве случаев можно обеспечить определенный уровень эффективности при помощи соответствующего повышения сложности решаемых задач. В этой связи, важной характеристикой параллельных вычислений становится соотношение необходимых темпов роста сложности расчетов и числа используемых процессоров.

Пусть  $E=const$  есть желаемый уровень эффективности выполняемых вычислений. Из выражения для эффективности можно получить

$$\frac{T_0}{T_1} = \frac{1-E}{E} \quad \text{или} \quad T_1 = KT_0, K = E/(1-E).$$

Порождаемую последним соотношением зависимость  $n=F(p)$  между сложностью решаемой задачи и числом процессоров обычно называют *функцией изоэффективности (isoefficiency function)* [72].

Покажем в качестве иллюстрации вывод функции изоэффективности для учебного примера суммирования числовых значений. В этом случае

$$T_0 = pT_p - T_1 = p((n/p) + \log_2 p) - n = p \log_2 p$$

и функция изоэффективности принимает вид

$$n = Kp \log_2 p.$$

Как результат, например, при числе процессоров  $p=16$  для обеспечения уровня эффективности  $E=0.5$  (т.е.  $K=1$ ) количество суммируемых значений должно быть не менее  $n=64$ . Или же, при увеличении числа процессоров с  $p$  до  $q$  ( $q>p$ ) для обеспечения пропорционального роста ускорения ( $S_q/S_p=(q/p)$ ) необходимо увеличить число суммируемых значений  $n$  в  $(q \log_2 q)/(p \log_2 p)$  раз.

## 2.8. Краткий обзор раздела

В разделе описывается модель вычислений в виде графа "операции-операнды", которая может использоваться для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач. В основу данной модели положен ациклический ориентированный граф, в котором вершины представляют операции, а дуги соответствуют зависимостям операций по данным. При наличии такого графа для определения параллельного алгоритма достаточно задать расписание, в соответствии с которым фиксируется распределение выполняемых операций по процессорам.

Представление вычислений при помощи моделей подобного вида позволяет получить аналитически ряд характеристик разрабатываемых параллельных алгоритмов, среди которых время выполнения, схема оптимального расписания, оценки максимально возможного быстродействия методов решения поставленных задач. Для возможности более простого построения теоретических оценок в разделе рассматривается понятие *паракомпьютера* как параллельной системы с неограниченным количеством процессоров.

Для оценки оптимальности разрабатываемых методов параллельных вычислений в разделе приводятся широко используемые в теории и практике параллельного программирования основные показатели качества - *ускорение (speedup)*, показывающее, во сколько раз быстрее осуществляется решение задач при

использовании нескольких процессоров, и **эффективность** (*efficiency*), которая характеризует долю времени реального использования процессоров вычислительной системы. Важной характеристикой разрабатываемых алгоритмов является **стоимость** (*cost*) вычислений, определяемая как произведение времени параллельного решения задачи и числа используемых процессоров.

Для демонстрации применимости рассмотренных моделей и методов анализа параллельных алгоритмов в разделе рассматривается задача нахождения частных сумм последовательности числовых значений. На данном примере отмечается проблема сложности распараллеливания последовательных алгоритмов, которые изначально не были ориентированы на возможность организации параллельных вычислений. Для выделения "скрытого" параллелизма показывается возможность преобразования исходной последовательной схемы вычислений и приводится получаемая в результате таких преобразований каскадная схема. На примере этой же задачи отмечается возможность введения избыточных вычислений для достижения большего параллелизма выполняемых расчетов.

В завершение раздела рассматривается вопрос построения оценок максимально достижимых значений показателей эффективности. Для получения таких оценок может быть использован закон Амдаля (*Amdahl*), позволяющий учесть существование последовательных (нераспараллелиемых) вычислений в методах решения задач. Закон Густавсона-Барсиса (*Gustafson-Barsis's law*) обеспечивает построение оценок ускорения масштабирования (*scaled speedup*), используемое для характеристики того, насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач. Для определения зависимости между сложностью решаемой задачи и числом процессоров, при соблюдении которой обеспечивается необходимый уровень эффективности параллельных вычислений, вводится понятие *функции изоэффективности* (*isoefficiency function*).

## 2.9. Обзор литературы

Дополнительная информация по моделированию и анализу параллельных вычислений может быть получена, например, в [8,10,44], полезная информация содержится также в [72,85].

Рассмотрение учебной задачи суммирования последовательности числовых значений было выполнено в [44].

Впервые закон Амдаля был изложен в работе [38]. Закон Густавсона-Барсиса был опубликован в работе [63]. Понятие функции изоэффективности было предложено в работе [61].

Систематическое изложение вопросов моделирования и анализа параллельных вычислений приводится в [102].

## 2.10. Контрольные вопросы

1. Как определяется модель "операция - операнды"?
2. Как определяется расписание для распределения вычислений между процессорами?
3. Как определяется время выполнения параллельного алгоритма?
4. Какое расписание является оптимальным?
5. Как определить минимально возможное время решения задачи?
6. Что понимается под паракомпьютером и для чего может оказаться полезным данное понятие?

7. Какие оценки следует использовать в качестве характеристики времени последовательного решения задачи?
8. Как определить минимально возможное время параллельного решения задачи по графу "операнды – операции"?
9. Какие зависимости могут быть получены для времени параллельного решения задачи при увеличении или уменьшении числа используемых процессоров?
10. При каком числе процессоров могут быть получены времена выполнения параллельного алгоритма, сопоставимые по порядку с оценками минимально возможного времени решения задачи?
11. Как определяются понятия ускорения и эффективности?
12. Возможно ли достижений сверхлинейного ускорения?
13. В чем состоит противоречивость показателей ускорения и эффективности?
14. Как определяется понятие стоимости вычислений?
15. В чем состоит понятие стоимостно-оптимального алгоритма?
16. В чем заключается проблема распараллеливания последовательного алгоритма суммирования числовых значений?
17. В чем состоит каскадная схема суммирования? С какой целью рассматривается модифицированный вариант данной схемы?
18. В чем различие показателей ускорения и эффективности для рассматриваемых вариантов каскадной схемы суммирования?
19. В чем состоит параллельный алгоритм вычисления всех частных сумм последовательности числовых значений?
20. Как формулируется закон Амдаля? Какой аспект параллельных вычислений позволяет учесть данный закон?
21. Какие предположения используются для обоснования закона Густавсона-Барсиса?
22. Как определяется функция изоэффективности?
23. Какой алгоритм является масштабируемым? Приведите примеры методов с разным уровнем масштабируемости.

## 2.11. Задачи и упражнения

1. Разработайте модель и выполните оценку показателей ускорения и эффективности параллельных вычислений:

- для задачи скалярного произведения двух векторов

$$y = \sum_{i=1}^N a_i b_i ,$$

- для задачи поиска максимального и минимального значений для заданного набора числовых данных

$$y_{\min} = \min_{1 \leq i \leq N} a_i, \quad y_{\max} = \max_{1 \leq i \leq N} a_i,$$

- для задачи нахождения среднего значения для заданного набора числовых данных

$$y = \frac{1}{N} \sum_{i=1}^N a_i .$$

2. Выполните в соответствии с законом Амдаля оценку максимально достижимого ускорения для задач п. 1.
3. Выполните оценку ускорения масштабирования для задач п.1.
4. Выполните построение функций изоэффективности для задач п. 1.
5. (\*) Разработайте модель и выполните полный анализ эффективности параллельных вычислений (ускорение, эффективность, максимально достижимое ускорение, ускорение масштабирования, функция изоэффективности) для задачи умножения матрицы на вектор.

Глава 3. Принципы разработки параллельных методов.....	1
3.1. Моделирование параллельных программ.....	3
3.2. Этапы разработки параллельных алгоритмов.....	5
3.2.1. Разделение вычислений на независимые части .....	5
3.2.2. Выделение информационных зависимостей .....	7
3.2.3. Масштабирование набора подзадач.....	8
3.2.4. Распределение подзадач между вычислительными элементами .....	9
3.3. Параллельное решение гравитационной задачи N тел .....	11
3.3.1. Разделение вычислений на независимые части .....	11
3.3.2. Выделение информационных зависимостей .....	11
3.3.3. Масштабирование и распределение подзадач по процессорам .....	11
3.4. Краткий обзор раздела .....	12
3.5. Обзор литературы .....	12
3.6. Контрольные вопросы .....	12
3.7. Задачи и упражнения.....	13

## Глава 3. Принципы разработки параллельных методов

Разработка алгоритмов (а в особенности методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. Для снижения сложности рассматриваемой темы оставим в стороне математические аспекты разработки и доказательства сходимости алгоритмов – эти вопросы в той или иной степени изучаются в ряде "классических" математических учебных курсов. Здесь же мы будем полагать, что вычислительные схемы решения задач, рассматриваемых далее в качестве примеров, уже известны<sup>1)</sup>. С учетом высказанных предположений последующие действия для определения эффективных способов организации параллельных вычислений могут состоять в следующем:

- Выполнить анализ имеющихся вычислительных схем и осуществить их разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга,
- Выделить для сформированного набора подзадач *информационные взаимодействия*, которые должны осуществляться в ходе решения исходной поставленной задачи,
- Определить необходимую (или доступную) для решения задачи *вычислительную систему* и выполнить *распределение* имеющего набора подзадач между процессорами системы.

---

<sup>1)</sup> Несмотря на то, что для многих научно-технических задач на самом деле известны не только последовательные, но и параллельные методы решения, данное предположение является, конечно, очень сильным, поскольку для новых возникающих задач, требующих для своего решения большого объема вычислений, процесс разработки алгоритмов составляет существенную часть всех выполняемых работ.

При самом общем рассмотрении понятно, что объем вычислений для каждого используемого процессора должен быть примерно одинаков – это позволит обеспечить равномерную вычислительную загрузку (*балансировку*) процессоров. Кроме того, также понятно, что распределение подзадач между процессорами должно быть выполнено таким образом, чтобы наличие информационных связей (*коммуникационных взаимодействий*) между подзадачами было минимальным.

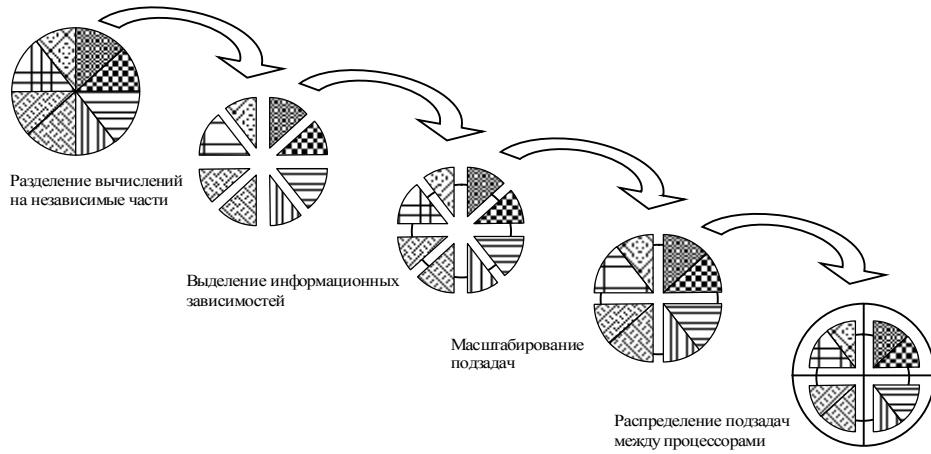


Рис. 3.1. Общая схема разработки параллельных алгоритмов

После выполнения всех перечисленных этапов проектирования можно оценить эффективность разрабатываемых параллельных методов – для этого обычно определяются значения показателей качества порождаемых параллельных вычислений (ускорение, эффективность, масштабируемость). По результатам проведенного анализа может оказаться необходимым повторение отдельных (в предельном случае всех) этапов разработки – следует отметить, что возврат к предшествующим шагам разработки может происходить на любой стадии проектирования параллельных вычислительных схем.

В этом отношении часто выполняемым дополнительным действием в приведенной выше схеме проектирования является корректировка состава сформированного множества задач после определения имеющегося количества процессоров – подзадачи могут быть укрупнены (*агрегированы*) при наличии малого числа процессоров или, наоборот, *детализированы* в противном случае. В целом, данные действия могут быть определены как *масштабирование* разрабатываемого алгоритма и выделены в качестве отдельного этапа проектирования параллельных вычислений.

Для применения получаемого в конечном итоге параллельного метода необходимо выполнить его программную реализацию и разделить разработанный программный код по вычислительным элементам компьютерной системы в соответствии с выбранной схемой распределения подзадач. Подготовленный программный код должен обеспечивать решение сформированного набора подзадач. Практически это может быть обеспечено, например, разработкой для решения каждой подзадачи отдельной программы, однако чаще всего создается программа, которая объединяет в себе все действия, необходимые для решения всех имеющихся подзадач. Такой объединенный программный код (*метапрограмма*) разрабатывается таким образом, чтобы программа в зависимости от управляющих параметров могла настраиваться на решение требуемой подзадачи (в качестве управляющего параметра может быть, например, номер вычислительного элемента). Для проведения вычислений подобная метапрограмма может копироваться для выполнения на все вычислительные элементы – такой подход используется, например, в технологии MPI для многопроцессорных вычислительных систем с распределенной памятью; выполняемые программы на разных

вычислительных элементах обычно именуются *процессами*. Метапрограмма может использоваться также и для порождения множества отдельных командных потоков – так, например, происходит в случае технологии OpenMP для многопроцессорных вычислительных систем с общей разделяемой памятью.

Для проведения вычислений параллельная программа запускается на выполнение, для реализации информационных взаимодействий параллельно выполняемые части программы (процессы или потоки) должны иметь в своем распоряжении средства обмена данными (*каналы передачи сообщений* для систем с распределенной памятью или *общие переменные* для систем с общей разделяемой памятью).

Каждый вычислительный элемент (процессор или ядро процессора) компьютерной системы обычно выделяется для решения одной единственной подзадачи, однако при наличии большого количества подзадач или использовании ограниченного числа вычислительных элементов это правило может не соблюдаться и, в результате, на вычислительных элементах может выполняться одновременно несколько параллельных частей программы (процессов или потоков). В частности, при разработке и начальной проверке параллельной программы для выполнения всех параллельных частей программы может использоваться один вычислительный элемент (при расположении на одном вычислительном элементе параллельные части программы выполняются в режиме разделения времени).

Следует отметить, что разработанная схема проектирования и реализации параллельных вычислений первоначально была предложена для вычислительных систем с распределенной памятью, когда необходимые информационные взаимодействия реализуются при помощи передачи сообщений по каналам связи между процессорами. Тем не менее, данная схема может быть использована без потери какой-либо эффективности параллельных вычислений и для разработки параллельных методов для систем с общей памятью – в этом случае механизмы передачи сообщений для обеспечения информационных взаимодействий просто заменяются операциями доступа к общим (разделяемым) переменным. Для снижения сложности излагаемого далее учебного материала *схема проектирования и реализации параллельных вычислений будет конкретизирована применительно к вычислительным системам с общей памятью*.

Важно отметить также, что для вычислительных систем с общей памятью активно пропагандируется и другой – "обратный" – способ разработки параллельных программ, когда за основу берется тот или иной последовательный прототип, который постепенно преобразуется к параллельному варианту (в частности, именно так предлагается использовать технологию OpenMP на начальном этапе освоения). Безусловно, такой подход позволяет достаточно быстро получить начальные варианты параллельных программ без значительных дополнительных усилий. Однако достаточно часто такая методика приводит к получению параллельных программ со сравнительно низкой эффективностью, и достижение максимально возможных показателей ускорения вычислений можно обеспечить только при изначальном проектировании параллельных вычислений на начальных этапах разработки методов решения поставленных задач.

### **3.1. Моделирование параллельных программ**

Рассмотренная схема проектирования и реализации параллельных вычислений дает способ понимания параллельных алгоритмов и программ. На стадии проектирования параллельный метод может быть представлен в виде *графа "подзадачи – информационные зависимости"*, который представляет собой не что иное, как укрупненное (агрегированное) представление графа информационных зависимостей (графа "операции-операнды" – см. Главу 2). Аналогично на стадии выполнения для

описания параллельной программы может быть использована модель в виде *графа "потоки – общие данные"*, в которой вместо подзадач используется понятие потоков, а информационные зависимости реализуются за счет использования общих данных. В дополнение, на этой модели может быть показано распределение потоков по вычислительным элементам компьютерной системы, если количество подзадач превышает число вычислительных элементов – см. рис. 3.2.

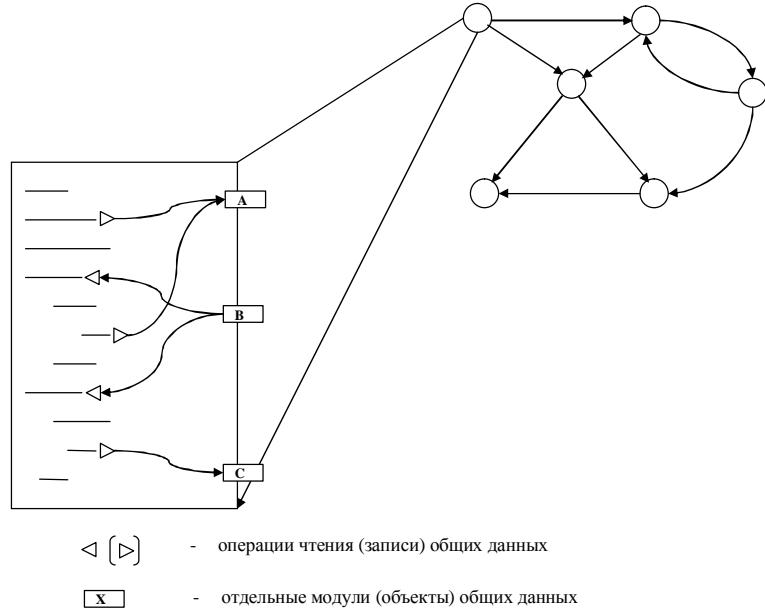


Рис. 3.2. Модель параллельной программы в виде графа "потоки – общие данные"

Использование двух моделей параллельных вычислений<sup>2)</sup> позволяет лучше разделить проблемы, которые проявляются при разработке параллельных методов. Первая модель – граф "подзадачи - информационные зависимости" – позволяет сосредоточиться на вопросах выделения подзадач одинаковой вычислительной сложности, обеспечивая при этом низкий уровень информационной зависимости между подзадачами. Вторая модель – граф "потоки – общие данные" – концентрирует внимание на вопросах распределения подзадач по вычислительным элементам и позволяет лучше анализировать эффективность разработанного параллельного метода и обеспечивает возможность более адекватного описания процесса выполнения параллельных вычислений.

Дадим дополнительные пояснения для используемых понятий в модели "потоки – общие данные":

- Под *потоком* в рамках данного учебного материала будем понимать логически выделенную с точки зрения операционной системы *последовательность команд*, которая может исполняться на одном из вычислительном элементе компьютерной системы и которая содержит ряд *операций доступа (чтения/записи)* к *общим данным* для организации информационного взаимодействия между выполняемыми потоками параллельной программы; все потоки параллельной программы имеют общее адресное пространство; в отличие от процессов операции создания и завершения потоков являются менее трудоемкими;

<sup>2)</sup> В работе Foster (1995) рассматривается только одна модель – модель "задача-канал" для описания параллельных вычислений, которая занимает некоторое промежуточное положение по сравнению с изложенными здесь моделями. Так, в модели "задача-канал" не учитывается возможность использования одного процессора для решения нескольких подзадач одновременно.

- *Общие данные* с логической точки зрения могут рассматриваться как *общий (разделяемый) между потоками ресурс*; общие данные могут использоваться параллельно выполняемыми потоками для чтения и записи значений.

Важно отметить, что для обеспечения корректности использование общих данных должно осуществляться в соответствии с правилами *взаимного исключения* (в каждый момент времени общие данные могут использоваться только одним потоком). При несоблюдении этих правил при использовании общих данных может иметь место ситуация *гонки потоков*, когда результат вычислений будет зависеть от взаимного соотношения темпа выполнения команд в потоках (такая ситуация возникает, например, при попытке записи значений общих данных одновременно несколькими потоками). С другой стороны, следует понимать также, что операции организации взаимоисключения могут приводить к задержкам (*блокировкам*) потоков.

Следует отметить важное достоинство рассмотренной модели "потоки – общие данные" – в этой модели проводится четкое разделение локальных (выполняемых на отдельном процессоре) вычислений и действий по организации информационного взаимодействия одновременно выполняемых потоков. Такой подход значительно снижает сложность анализа эффективности параллельных методов и существенно упрощает проблемы разработки параллельных программ.

## 3.2. Этапы разработки параллельных алгоритмов

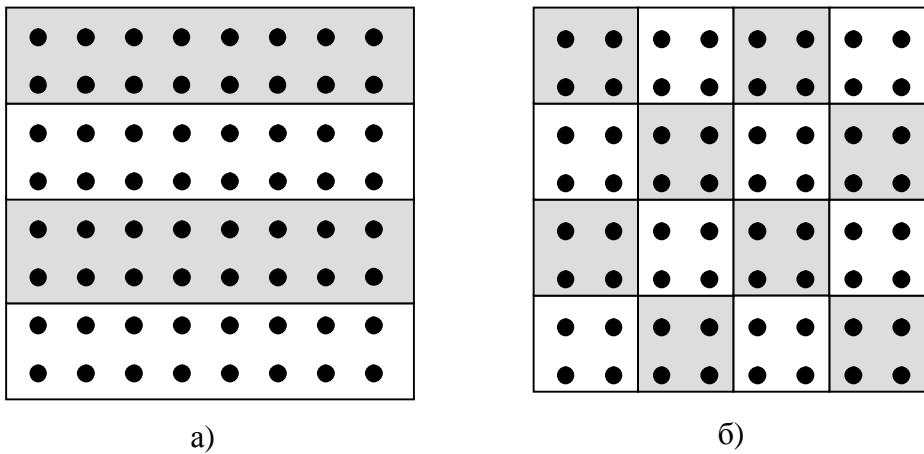
Рассмотрим более подробно изложенную выше методику разработки параллельных алгоритмов. В значительной степени данная методика опирается на подход, впервые разработанный в [54], и, как отмечалось ранее, включает этапы выделения подзадач, определения информационных зависимостей, масштабирования и распределения подзадач по процессорам вычислительной системы (см. рис. 3.1). Для демонстрации приводимых рекомендаций далее будет использоваться учебная задача поиска максимального значения среди элементов матрицы  $A$  (такая задача возникает, например, при численном решении систем линейных уравнений для определения ведущего элемента метода Гаусса):

$$y = \max_{1 \leq i, j \leq N} a_{ij}.$$

Такая задача носит полностью иллюстративный характер, и после рассмотрения этапов разработки в оставшейся части раздела будет приведен более полный пример использования данной методики для разработки параллельных алгоритмов. Кроме того, данная схема разработки будет применена и при изложении всех далее рассматриваемых методов параллельных вычислений.

### 3.2.1. Разделение вычислений на независимые части

Выбор способа разделения вычислений на независимые части основывается на анализе вычислительной схемы решения исходной задачи. Требования, которым должен удовлетворять выбираемый подход, обычно состоят в обеспечении равного объема вычислений в выделяемых подзадачах и минимума информационных зависимостей между этими подзадачами (при прочих равных условиях нужно отдавать предпочтение редким операциям передачи большего размера сообщений по сравнению с частыми пересылками данных небольшого объема). В общем случае, проведение анализа и выделение задач представляет собой достаточно сложную проблему – ситуацию помогает разрешить существование двух часто встречающихся типов вычислительных схем:



а)

б)

Рис. 3.3. Разделение данных для матрицы  $A$ :  
а) ленточная схема, б) блочная схема

- Для большого класса задач вычисления сводятся к выполнению однотипной обработки большого набора данных – к такому виду задач относятся, например, матричные вычисления, численные методы решения уравнений в частных производных и др. В этом случае говорят, что существует *параллелизм по данным*, и выделение подзадач сводится к разделению имеющихся данных. Так, например, для нашей учебной задачи поиска максимального значения при формировании подзадач исходная матрица  $A$  может быть разделена на отдельные строки (или последовательные группы строк) – *ленточная схема* разделения данных (см. рис. 3.3) или на прямоугольные наборы элементов – *блочная схема* разделения данных. Для большого количества решаемых задач разделение вычислений по данным приводит к порождению одно-, двух- и трех- мерных наборов подзадач, для которых информационные связи существуют только между ближайшими соседями (такие схемы обычно называются *сетками* или *решетками*),

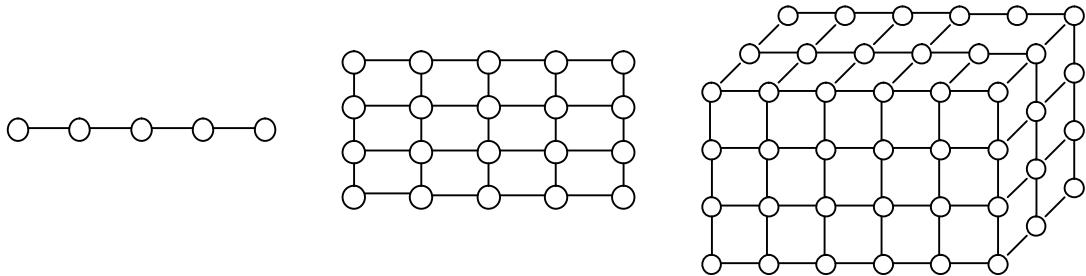


Рис. 3.4. Регулярные одно-, двух- и трех- мерные структуры базовых подзадач  
после декомпозиции данных

- Для другой части задач вычисления могут состоять в выполнении разных операций над одним и тем же набором данных – в этом случае говорят о существовании *функционального параллелизма* (в качестве примеров можно привести задачи обработки последовательности запросов к информационным базам данных, вычисления с одновременным применением разных алгоритмов расчета и т.п.). Очень часто функциональная декомпозиция может быть использована для организации конвейерной обработки данных (так, например, при выполнении каких-либо преобразований данных вычисления могут быть сведены к функциональной последовательности ввода, обработки и сохранения данных).

Важный вопрос при выделении подзадач состоит в выборе нужного уровня декомпозиции вычислений. Формирование максимально возможного количества

подзадач обеспечивает использование предельно достижимого уровня параллелизма решаемой задачи, однако затрудняет анализ параллельных вычислений. Использование при декомпозиции вычислений только достаточно "крупных" подзадач приводит к ясной схеме параллельных вычислений, однако может затруднить эффективное использование достаточно большого количества процессоров. Возможное разумное сочетание этих двух подходов может состоять в использовании в качестве конструктивных элементов декомпозиции только тех подзадач, для которых методы параллельных вычислений являются известными. Так, например, при анализе задачи матричного умножения в качестве подзадач можно использовать методы скалярного произведения векторов или алгоритмы матрично-векторного произведения. Подобный промежуточный способ декомпозиции вычислений позволит обеспечить и простоту представления вычислительных схем, и эффективность параллельных расчетов. Выбираемые подзадачи при таком подходе будем именовать далее *базовыми*, которые могут быть *элементарными* (неделимыми), если не допускают дальнейшего разделения, или *составными* в противном случае.

Для рассматриваемой учебной задачи достаточный уровень декомпозиции может состоять, например, в разделении матрицы  $A$  на множество отдельных строк и получении на этой основе набора подзадач поиска максимальных значений в отдельных строках; порождаемая при этом структура информационных связей соответствует линейному графу – см. рис. 3.5.

Для оценки корректности этапа разделения вычислений на независимые части можно воспользоваться контрольным списком вопросов, предложенных в [54]:

- Выполненная декомпозиция не увеличивает объем вычислений и необходимый объем памяти?
- Возможна ли при выбранном способе декомпозиции равномерная загрузка всех имеющихся вычислительных элементов компьютерной системы?
- Достаточно ли выделенных частей процесса вычислений для эффективной загрузки имеющихся вычислительных элементов (с учетом возможности увеличения их количества)?

### 3.2.2. Выделение информационных зависимостей

При наличии вычислительной схемы решения задачи после выделения базовых подзадач определение информационных зависимостей между подзадачами обычно не вызывает больших затруднений. При этом, однако, следует отметить, что на самом деле этапы выделения подзадач и информационных зависимостей достаточно сложно поддаются разделению. Выделение подзадач должно происходить с учетом возникающих информационных связей; после анализа объема и частоты необходимых информационных обменов между подзадачами может потребоваться повторение этапа разделения вычислений.

При проведении анализа информационных зависимостей между подзадачами следует различать (предпочтительные формы информационного взаимодействия выделены подчеркиванием):

- *Локальные* и *глобальные* схемы информационного взаимодействия – для локальных схем в каждый момент времени информационная зависимость существует только между небольшим числом подзадач (располагаемых, как правило, на соседних вычислительных элементах), для глобальных зависимостей информационное взаимодействие имеет место для всех подзадач,
- *Структурные* и *произвольные* способы взаимодействия – для структурных способов организация взаимодействий приводит к формированию некоторых

стандартных схем коммуникации (например, в виде кольца, прямоугольной решетки и т.д.), для произвольных структур взаимодействия схема информационных зависимостей не носит характер какой-либо однородности,

- *Статические* или *динамические* схемы информационной зависимости – для статических схем моменты и участники информационного взаимодействия фиксируются на этапах проектирования и разработки параллельных программ, для динамического варианта взаимодействия структура зависимостей определяется в ходе выполняемых вычислений.

Как уже отмечалось в предыдущем пункте, для учебной задачи поиска максимального значения при использовании в качестве базовых элементов подзадач поиска максимальных значений в отдельных строках исходной матрицы  $A$  структура информационных связей имеет вид, представленный на рис. 3.5.

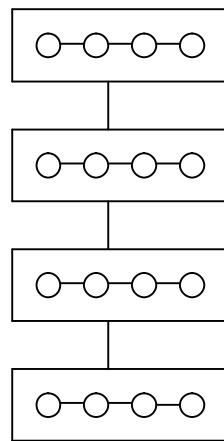


Рис. 3.5. Структура информационных связей учебной задачи

Как и ранее, для оценки правильности этапа выделения информационных зависимостей можно воспользоваться контрольным списком вопросов, предложенных в [54]:

- Соответствует ли вычислительная сложность подзадач интенсивности их информационных взаимодействий?
- Является ли одинаковой интенсивность информационных взаимодействий для разных подзадач?
- Является ли схема информационного взаимодействия локальной?
- Не препятствует ли выявленная информационная зависимость параллельному решению подзадач?

### 3.2.3. Масштабирование набора подзадач

Масштабирование разработанной вычислительной схемы параллельных вычислений проводится в случае, если количество имеющихся подзадач отличается от числа планируемых к использованию вычислительных элементов. Для сокращения количества подзадач необходимо выполнить укрупнение (*агрегацию*) вычислений. Применяемые здесь правила совпадают с рекомендациями начального этапа выделения подзадач – определяемые подзадачи, как и ранее, должны иметь одинаковую вычислительную сложность, а объем и интенсивность информационных взаимодействий между подзадачами должны оставаться на минимально-возможном уровне. Как результат, первыми претендентами на объединение являются подзадачи с высокой степенью информационной взаимозависимости.

При недостаточном количестве имеющегося набора подзадач для загрузки всех доступных к использованию вычислительных элементов необходимо выполнить детализацию (*декомпозицию*) вычислений. Как правило, проведение подобной декомпозиции не вызывает каких-либо затруднений, если для базовых задач методы параллельных вычислений являются известными.

Выполнение этапа масштабирования вычислений должно свестись, в конечном итоге, к разработке правил агрегации и декомпозиции подзадач, которые должны параметрически зависеть от числа вычислительных элементов, применяемых для вычислений.

Для рассматриваемой учебной задачи поиска максимального значения агрегация вычислений может состоять в объединении отдельных строк в группы (ленточная схема разделения матрицы – см. рис. 3.3а), при декомпозиции подзадач строки исходной матрицы  $A$  могут разбиваться на несколько частей (блоков).

Список контрольных вопросов, предложенный в [54] для оценки правильности этапа масштабирования, выглядит следующим образом:

- Не ухудшится ли локальность вычислений после масштабирования имеющегося набора подзадач?
- Имеют ли подзадачи после масштабирования одинаковую вычислительную и коммуникационную сложность?
- Соответствует ли количество задач числу имеющихся вычислительных элементов?
- Зависят ли параметрически правила масштабирования от количества вычислительных элементов?

### **3.2.4. Распределение подзадач между вычислительными элементами**

Распределение подзадач между вычислительными элементами является завершающим этапом разработки параллельного метода. Надо отметить, что управление распределением нагрузки для процессоров возможно только для вычислительных систем с распределенной памятью, для мультипроцессоров (систем с общей памятью) распределение нагрузки обычно выполняется операционной системой автоматически. Кроме того, данный этап распределения подзадач между процессорами является избыточным, если количество подзадач совпадает с числом имеющихся вычислительных элементов.

Основной показатель успешности выполнения данного этапа – *эффективность использования вычислительных элементов*, определяемая как относительная доля времени, в течение которого вычислительные элементы использовались для вычислений, связанных с решением исходной задачи. Пути достижения хороших результатов в этом направлении остаются прежними – как и ранее, необходимо обеспечить равномерное распределение вычислительной нагрузки между вычислительными элементами и минимизировать количество информационных взаимодействий, существующих между ними. Точно так же, как и на предшествующих этапах проектирования, оптимальное решение проблемы распределения подзадач между вычислительными элементами основывается на анализе информационной связности графа "подзадачи – информационные зависимости". Так, в частности, подзадачи, между которыми имеются интенсивные информационные взаимодействия, целесообразно размещать на вычислительных элементах (ядрах) одного и того же процессора.

Следует отметить, что требование минимизации информационных обменов между вычислительными элементами может противоречить условию равномерной загрузки

процессоров вычислительной системы. Так, мы можем разместить все подзадачи на одном процессоре и полностью устраниТЬ межпроцессорную передачу данных, однако, понятно, загрузка большинства процессоров в этом случае будет минимальной.

Для нашей учебной задачи поиска максимального значения распределение подзадач между вычислительными элементами не вызывает каких-либо затруднений – размещение подзадач может быть выполнено непосредственно операционной системой.

Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений может изменяться в ходе решения задачи. Причиной этого могут быть, например, неоднородные сетки при решении уравнений в частных производных, разреженность матриц и т.п.<sup>3)</sup>. Кроме того, используемые на этапах проектирования оценки вычислительной сложности решения подзадач могут иметь приближенный характер и, наконец, количество подзадач может изменяться в ходе вычислений. В таких ситуациях может потребоваться перераспределение базовых подзадач между процессорами уже непосредственно в процессе выполнения параллельной программы (или, как обычно говорят, придется выполнить *динамическую балансировку* вычислительной нагрузки). Данные вопросы являются одними из наиболее сложных (и наиболее интересных) в области параллельных вычислений – к сожалению, рассмотрение данных вопросов выходит за рамки данного учебного материала (дополнительная информация может быть получена, например, в [47,99]).

В качестве примера дадим краткую характеристику широко используемого способа динамического управления распределением вычислительной нагрузки, обычно именуемого *схемой "менеджер - исполнитель"* (*manager-worker scheme*). При использовании данного подхода предполагается, что подзадачи могут возникать и завершаться в ходе вычислений, при этом информационные взаимодействия между подзадачами либо полностью отсутствует, либо минимальны. В соответствии с рассматриваемой схемой для управления распределением нагрузки в системе выделяется отдельный вычислительный элемент-менеджер, которому доступна информация обо всех имеющихся подзадачах. Остальные вычислительные элементы системы являются исполнителями, которые для получения вычислительной нагрузки обращаются к менеджеру. Порождаемые в ходе вычислений новые подзадачи передаются обратно менеджеру и могут быть получены для решения при последующих обращениях исполнителей. Завершение вычислений происходит в момент, когда исполнители завершили решение всех переданных им подзадач, а менеджер не имеет каких-либо вычислительных работ для выполнения.

Предложенный в [54] перечень контрольных вопросов для проверки этапа распределения подзадач состоит в следующем:

- Не приводит ли распределение нескольких задач на один процессор к росту дополнительных вычислительных затрат?
- Существует ли необходимость динамической балансировки вычислений?
- Не является ли вычислительный элемент-менеджер "узким" местом при использовании схемы "менеджер-исполнитель"?

---

<sup>3)</sup> Можно отметить, что даже для нашей простой учебной задачи может наблюдаться различная вычислительная сложность сформированных базовых задач. Так, например, количество операций при поиске максимального значения для строки, в которой максимальное значение имеет первый элемент, и строки, в которой значения являются упорядоченными по возрастанию, будет различаться в два раза.

### **3.3. Параллельное решение гравитационной задачи $N$ тел**

Рассмотрим более сложный пример задачи для демонстрации приведенной выше схемы проектирования и реализации параллельных вычислений.

Многие задачи в области физики сводятся к операциям обработки данных для каждой пары объектов имеющейся физической системы. Такой задачей является, в частности, проблема, широко известная в литературе как *гравитационная задача  $N$  тел* (или просто *задача  $N$  тел*) – см., например, [39] В самом общем виде, задача может быть описана следующим образом.

Пусть дано большое количество тел (планет, звезд и т.д.), для каждого из которых известна масса, начальное положение и скорость. Под действием гравитации положение тел меняется, и требуемое решение задачи состоит в моделировании динамики изменения системы  $N$  тел на протяжении некоторого задаваемого интервала времени. Для проведения такого моделирования заданный интервал времени обычно разбивается на временные отрезки небольшой длительности и далее на каждом шаге моделирования вычисляются силы, действующие на каждое тело, а затем обновляются скорости и положения тел.

Очевидный алгоритм решения задачи  $N$  тел состоит в рассмотрении на каждом шаге моделирования всех пар объектов физической системы и выполнении для каждой получаемой пары всех необходимых расчетов.

#### **3.3.1. Разделение вычислений на независимые части**

Выбор способа разделения вычислений не вызывает каких-либо затруднений – очевидный подход состоит в выборе в качестве базовой подзадачи всего набора вычислений, связанных с обработкой данных одного какого-либо тела физической системы.

#### **3.3.2. Выделение информационных зависимостей**

Выполнение вычислений, связанных с каждой подзадачей, становится возможным только в случае, когда в подзадачах имеются данные (положение и скорости передвижения) обо всех телах имеющейся физической системы. Как результат, перед началом каждой итерации моделирования каждая подзадача должна получить все необходимые сведения от всех других подзадач системы. Такая процедура информационного взаимодействия обычно именуется *операцией сбора данных (single-node gather)*. В рассматриваемом алгоритме данная операция должна быть выполнена для каждой подзадачи – такой вариант информационного взаимодействия часто именуется как *операция обобщенного сбора данных (multi-node gather or all gather)*.

#### **3.3.3. Масштабирование и распределение подзадач по процессорам**

Как правило, число тел физической системы  $N$  значительно превышает количество вычислительных элементов  $p$ . Как результат, рассмотренные ранее подзадачи следует укрупнить, объединив в рамках одной подзадачи вычисления для группы ( $N/p$ ) тел – после проведения подобной агрегации число подзадач и количество вычислительных элементов будет совпадать.

Учитывая характер имеющихся информационных зависимостей распределение подзадач между вычислительными элементами может быть непосредственно операционной системой.

### **3.4. Краткий обзор главы**

В главе была рассмотрена методика разработки параллельных алгоритмов, предложенная в [54]. Данная методика включает этапы выделения подзадач, определения информационных зависимостей, масштабирования и распределения подзадач по вычислительным элементам компьютерной системы. При применении методики предполагается, что вычислительная схема решения рассматриваемой задачи уже является известной. Основные требования, которые должны быть обеспечены при разработке параллельных алгоритмов, состоят в обеспечении равномерной загрузки вычислительных элементов при низком информационном взаимодействии сформированного множества подзадач.

Для описания получаемых в ходе разработки вычислительных параллельных схем рассмотрены две модели. Первая из них – модель "подзадачи – информационные зависимости" может быть использована на стадии проектирования параллельных алгоритмов, вторая модель "потоки – общие данные" может быть применена на стадии реализации методов в виде параллельных программ.

В завершение раздела показывается применение рассмотренной методики разработки параллельных алгоритмов на примере решения гравитационной задачи  $N$  тел.

### **3.5. Обзор литературы**

Рассмотренная в разделе методика разработки параллельных алгоритмов впервые была предложена в [54]. В этой работе изложение методики проводится более детально; кроме того, в работе содержится несколько примеров использования методики для разработки параллельных методов для решения ряда вычислительных задач.

Полезной при рассмотрении вопросов проектирования и разработки параллельных алгоритмов может оказаться также работа [85].

Гравитационная задача  $N$  тел более подробно рассматривается в [39].

### **3.6. Контрольные вопросы**

1. В чем состоят исходные предположения для возможности применения рассмотренной в разделе методики разработки параллельных алгоритмов?
2. Каковы основные этапы проектирования и разработки методов параллельных вычислений?
3. Как определяется модель "подзадачи-информационные зависимости"?
4. Как определяется модель "потоки-общие данные"?
5. Какие основные требования должны быть обеспечены при разработке параллельных алгоритмов?
6. В чем состоят основные действия на этапе выделения подзадач?
7. Каковы основные действия на этапе определения информационных зависимостей?
8. В чем состоят основные действия на этапе масштабирования имеющегося набора подзадач?
9. В чем состоят основные действия на этапе распределения подзадач по вычислительным элементам компьютерной системы?
10. Как происходит динамическое управление распределением вычислительной нагрузки при помощи схемы "менеджер - исполнитель"?

11. Какой метод параллельных вычислений был разработан для решения гравитационной задачи  $N$  тел?

### 3.7. Задачи и упражнения

1. Разработайте схему параллельных вычислений, используя рассмотренную в разделе методику проектирования и разработки параллельных методов:

- для задачи поиска максимального значения среди минимальных элементов строк матрицы (такая задача имеет место для решения матричных игр)

$$y = \max_{1 \leq i \leq N} \min_{1 \leq j \leq N} a_{ij},$$

(обратите особое внимание на ситуацию, когда число процессоров превышает порядок матрицы, т.е.  $p > N$ ),

- для задачи вычисления определенного интеграла с использованием метода прямоугольников

$$y = \int_a^b f(x) dx \approx h \sum_{i=0}^{N-1} f_i, \quad f_i = f(x_i), \quad x_i = i h, \quad h = (b - a) / N.$$

(описание методов интегрирования дано, например, в [68]),

- Разработайте схему параллельных вычислений для задачи умножения матрицы на вектор, используя рассмотренную в разделе методику проектирования и разработки параллельных методов.

<b>Глава 4. Основы параллельного программирования .....</b>	2
<b>4.1. Основные понятия.....</b>	2
4.1.1. Концепция процесса.....	2
4.1.2. Определение потока .....	3
4.1.3. Понятие ресурса .....	3
4.1.4. Организация параллельных программ как системы потоков.....	4
<b>4.2. Взаимодействие и взаимоисключение потоков.....</b>	7
<b>4.2.1. Разработка алгоритма взаимоисключения .....</b>	7
<i>Вариант 1 – Жесткая синхронизация .....</i>	8
<i>Вариант 2 – Потеря взаимоисключения .....</i>	8
<i>Вариант 3 – Возможность взаимоблокировки .....</i>	9
<i>Вариант 4 – Бесконечное откладывание .....</i>	10
<i>Алгоритм Деккера .....</i>	10
4.2.2. Семафоры.....	11
4.2.3. Мониторы.....	12
<b>4.3. Синхронизация потоков.....</b>	15
4.3.1. Условные переменные .....	15
4.3.2. Барьерная синхронизация .....	15
<b>4.4. Взаимоблокировка потоков.....</b>	16
4.4.1. Модель программы в виде графа "поток-ресурс" .....	17
4.4.2. Описание возможных изменений состояния программы .....	18
4.4.3. Обнаружение и исключение тупиков.....	19
<b>4.5. Классические задачи синхронизации .....</b>	19
4.5.1. Задача "Производители-Потребители" .....	20
4.5.2. Задача "Читатели-Писатели" .....	21
4.5.3. Задача "Обедающие философы" .....	22
4.5.4. Задача "Спящий парикмахер" .....	24
<b>4.6. Методы повышения эффективности параллельных программ.....</b>	26
4.6.1. Оптимизация количества потоков .....	26
4.6.2. Минимизация взаимодействия потоков .....	27
4.6.3. Оптимизация работы с памятью .....	27
<i>Обеспечение однозначности кэш-памяти.....</i>	28
<i>Уменьшение миграции потоков между ядрами/процессорами .....</i>	28
<i>Устранение эффекта ложного разделения данных .....</i>	28
4.6.4. Использование потоко-ориентированных библиотек.....	28
<b>4.7. Краткий обзор раздела .....</b>	29
<b>4.8. Обзор литературы.....</b>	29
<b>4.9. Контрольные вопросы .....</b>	30
<b>4.10. Задачи и упражнения.....</b>	31

## Глава 4.

### Основы параллельного программирования

#### 4.1. Основные понятия

Основой для разработки параллельных программ, реализующих параллельные методы решения задач, является понятия *процесса* и *потока*. Рассмотрение программы в виде набора процессов/потоков, выполняемых параллельно на разных процессорах или на одном процессоре в режиме разделения времени, позволяет сконцентрироваться на рассмотрении проблем организации *взаимодействия параллельных частей программы*, определить моменты и способы обеспечения *синхронизации* и *взаимоисключения* *процессов/потоков*, изучить условия возникновения или доказать отсутствие *тупиков* в ходе выполнения программ (ситуаций, в которых все или часть параллельных участков программы не могут быть выполнены при любых вариантах продолжения вычислений).

##### 4.1.1. Концепция процесса

Понятие *процесса* является одним из основополагающих в теории и практике параллельного программирования. В научно-технической литературе дается ряд определений процесса, но в целом большинство определений сводится к пониманию процесса как "*некоторой последовательности команд, претендующей наравне с другими процессами программы на использование процессора для своего выполнения*".

Конкретизация понятия процесса зависит от целей исследования параллельных программ. Для анализа проблем организации взаимодействия процессов процесс можно рассматривать как последовательность команд

$$p_n = (i_1, i_2, \dots, i_n)$$

(для простоты изложения материала будем предполагать, что процесс описывается единственной командной последовательностью). Динамика развития процесса определяется моментами времен начала выполнения команд

$$t(p_n) = t_p = (t_1, t_2, \dots, t_n),$$

где  $t_j$ ,  $1 \leq j \leq n$ , есть время начала выполнения команды  $j$ . Последовательность  $t_p$  представляет временную *траекторию* развития процесса. Предполагая, что команды процесса исполняются строго последовательно, в ходе своей реализации не могут быть приостановлены (т.е. являются неделимыми) и имеют одинаковую длительность выполнения, равную 1 (в тех или иных временных единицах), получим, что моменты времени траектории процесса должны удовлетворять соотношениям

$$\forall i, \quad 1 \leq i < n \Rightarrow t_{i+1} \geq t_i + 1.$$

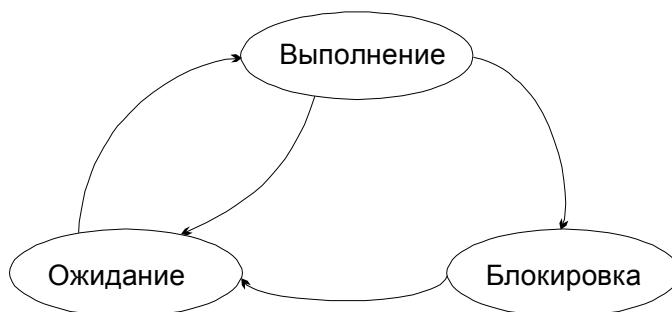


Рис. 4.1. Диаграмма переходов процесса из состояния в состояние

Равенство  $t_{i+1} = t_i + 1$  достигается, если для выполнения процесса выделен процессор и после завершения очередной команды процесса сразу же начинается выполнение следующей команды. В этом случае говорят, что процесс является *активным* и находится в *состоянии выполнения*. Соотношение  $t_{i+1} > t_i + 1$  означает, что после выполнения очередной команды процесс *приостановлен* и ожидает возможности для своего продолжения. Данная приостановка может быть вызвана необходимостью разделения использования единственного процессора между одновременно исполняемыми процессами. В этом случае приостановленный процесс находится в *состоянии ожидания* момента предоставления процессора для своего выполнения. Кроме того, приостановка процесса может быть вызвана и временной неготовностью процесса к дальнейшему выполнению (например, процесс может быть продолжен только после завершения операции ввода-вывода данных). В подобных ситуациях говорят, что процесс является *блокированным* и находится в *состоянии блокировки*.

В ходе своего выполнения состояние процесса может многократно изменяться; возможные варианты смены состояний показаны на диаграмме переходов рис. 4.1.

#### 4.1.2. Определение потока

Как уже отмечалось ранее, понятие процесса является чрезвычайно важным для организации вычислений на ЭВМ. Выполняемые программы с точки зрения операционной системы представляют собой процессы, которые параллельно выполняются, взаимодействуют между собой и конкурируют за использование процессоров вычислительной системы. Вместе с этим, понятие процесса является несколько "тяжеловесным" – создание процесса, переключение процессоров на использование других процессов и выполнение других подобных действий занимает достаточно много процессорного времени. Кроме того, процессы выполняются в разных адресных пространствах и, как результат, организация их взаимодействия требует определенных усилий.

Все выше отмеченные моменты приводят к необходимости определения *потока* (*thread*) как более простой альтернативы понятию процесса. Поток (точно также как и процесс) можно представлять как последовательность команд программы, которая может претендовать на использование процессора вычислительной системы для своего выполнения. Однако – в отличие от процесса – потоки одной и той же программы работают в общем адресном пространстве и, тем самым, разделяют данные программы. Следует отметить, что общность данных, с одной стороны, существенно упрощает организацию взаимодействия потоков (результат, вычисленный одним потоком, сразу становится доступным всем остальным потокам программы), но, с другой стороны, требует соблюдения определенных правил использования разделяемых данных – данный аспект обсуждается далее в п. 4.1.4.

Процессы и потоки в равной степени используются для организации вычислений. Процессы применяются для представления отдельных программ (заданий для операционной системы) и для формирования многопроцессных программ, исполняемых на вычислительных системах с распределенной памятью. Потоки используются для представления программ как множества частей (потоков), которые могут выполняться независимо друг от друга (параллельно).

Дополнительная информация по более детальному представлению процессов и потоков может быть получена в [5,27,31,33].

#### 4.1.3. Понятие ресурса

Понятие *ресурса* обычно используется для обозначения любых объектов вычислительной системы, которые могут быть использованы процессом для своего

выполнения. В качестве ресурса может рассматриваться процесс, память, программы, данные и т.п. По характеру использования могут различаться следующие категории ресурсов:

- *выделяемые* (монопольно используемые, неперераспределяемые) ресурсы характеризуются тем, что выделяются процессам в момент их возникновения и освобождаются только в момент завершения процессов; в качестве такого ресурса может рассматриваться, например, устройство чтения на магнитных лентах;
- *повторно распределяемые ресурсы* отличаются возможностью динамического запрашивания, выделения и освобождения в ходе выполнения процессов (таковым ресурсом является, например, оперативная память);
- *разделяемые* ресурсы, особенность которых состоит в том, что они постоянно остаются в общем использовании и выделяются процессам для использования в режиме разделения времени (как, например, процессор, разделяемые файлы и т.п.);
- *многократно используемые* (реентерабельные) ресурсы выделяются возможностью одновременного использования несколькими процессами (что может быть обеспечено, например, при неизменяемости ресурса при его использовании; в качестве примеров таких ресурсов могут рассматриваться реентерабельные программы, файлы, используемые только для чтения и т.д.).

Следует отметить, что тип ресурса определяется не только его конкретными характеристиками, но и зависит от применяемого способа использования. Так, например, оперативная память может рассматриваться как повторно распределяемый, так и разделяемый ресурс; использование программ может быть организовано в виде ресурса любого рассмотренного типа.

#### **4.1.4. Организация параллельных программ как системы потоков**

Ориентируясь в данном учебном издании на проблемы разработки параллельных программ для вычислительных систем с общей памятью, далее все рассматриваемые вопросы будут даваться на примере потоков. Подобный подход позволяет снизить сложность изучения излагаемого учебного материала, но при этом следует понимать, что данный материал является общим для параллельного программирования в целом.

Итак, понятие потока может быть использовано в качестве основного конструктивного элемента для построения параллельных программ в виде совокупности взаимодействующих потоков. Такая агрегация программы позволяет получить более компактные (поддающиеся анализу) вычислительные схемы реализуемых методов, скрыть при выборе способов распараллеливания несущественные детали программной реализации, обеспечивает концентрацию усилий на решение основных проблем параллельного функционирования программ.

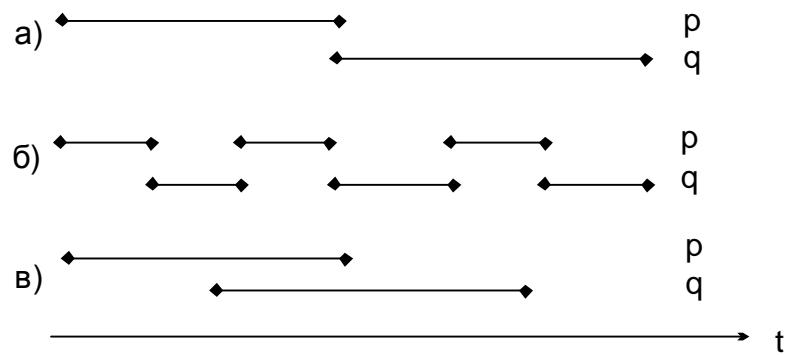


Рис. 4.2. Варианты взаиморасположения траекторий одновременно исполняемых потоков (отрезки линий изображают фрагменты командных последовательностей потоков)

Существование нескольких одновременно выполняемых потоков приводит к появлению дополнительных соотношений, которые должны выполняться для величин временных траекторий потоков. Возможные типовые варианты таких соотношений на примере двух потоков  $p$  и  $q$  состоят в следующем (см. рис. 4.2):

- выполнение потоков осуществляется строго последовательно, т.е. поток  $q$  начинает свое выполнение только после полного завершения потока  $p$  (*однопрограммный режим работы ЭВМ* – см. рис. 4.2а),
- выполнение потоков может осуществляться одновременно, но в каждый момент времени могут исполняться команды только какого либо одного потока (*режим разделения времени* или *многопрограммный режим работы ЭВМ* – см. рис. 4.2б),
- *параллельное выполнение* потоков, когда одновременно могут выполняться команды нескольких потоков (данный режим исполнения потоков осуществим только при наличии в вычислительной системе нескольких процессоров - см. рис. 4.2в).

Приведенные варианты взаиморасположения траекторий потоков определяются не требованиями необходимых функциональных взаимодействий потоков, а являются лишь следствием технической реализации одновременной работы нескольких потоков. С другой стороны, возможность чередования по времени командных последовательностей разных потоков следует учитывать при разработке многопоточной программы. Рассмотрим для примера два потока с идентичным программным кодом.

<b>Поток 1</b>	<b>Поток 2</b>
$N = N + 1$	$N = N + 1$
печать $N$	печать $N$

Пусть начальное значение переменной  $N$  равно 1. Тогда при последовательном исполнении поток 1 напечатает значение 2, а поток 2 – значение 3. Однако возможна и другая последовательность исполнения потоков в режиме разделения времени (с учетом того, что сложение  $N = N + 1$  выполняется при помощи нескольких машинных команд)

<b>Время</b>	<b>Поток 1</b>	<b>Поток 2</b>
1	Чтение $N(1)$	
2		Чтение $N(1)$
3		Прибавление 1 (2)
4	Прибавление 1 (2)	
5	Запись $N(2)$	
6	Печать $N(2)$	
7		Запись $N(2)$
8		Печать $N(2)$

(в скобках для каждой команды указывается значение переменной  $N$ ). Как следует из приведенного примера, результат одновременного выполнения нескольких потоков, если не предпринимать специальных мер, может зависеть от порядка исполнения команд. Ситуация, когда два или более потоков используют разделяемый ресурс и конечный результат зависит от соотношения скоростей потоков, называется *состязанием* или *гонками* (*race conditions*).

Выполним анализ возможных командных последовательностей, которые могут получаться для программ, образованных в виде набора потоков. Рассмотрим для простоты два потока

$$p_n = (i_1, i_2, \mathbf{L}, i_n), q_m = (j_1, j_2, \mathbf{L}, j_m).$$

Командная последовательность программы образуется чередованием команд отдельных потоков и, тем самым, имеет вид:

$$r_s = (l_1, l_2, \mathbf{L}, l_s), s = n + m.$$

Фиксация способа образования последовательности  $r_s$  из команд отдельных потоков может быть обеспечена при помощи характеристического вектора

$$x_s = (c_1, c_2, \mathbf{L}, c_s),$$

в котором следует положить  $c_k = p_n$ , если команда  $l_k$  получена из потока  $p_n$  (иначе  $c_k = q_m$ ). Порядок следования команд потоков в  $r_s$  должен соответствовать порядку расположения этих команд в исходных потоках

$$\forall u, v : (u < v), (c_u = c_v = p_n) \Rightarrow p_n(l_u) < p_n(l_v),$$

где  $p_n(l_k)$  есть команда потока  $p_n$ , соответствующая команде  $l_k$  в  $r_s$ .

С учетом введенных обозначений, под программой, образованной из потоков  $p_n$  и  $q_m$ , можно понимать множество всех возможных командных последовательностей

$$R_s = \{< r_s, x_s >\}.$$

Данный подход позволяет рассматривать программу так же, как некоторый обобщенный (*агрегированный*) поток, получаемый путем параллельного объединения составляющих потоков

$$R_s = p_n \otimes q_m.$$

Выделенные особенности одновременного выполнения нескольких потоков могут быть сформулированы в виде ряда **принципиальных положений**, которые должны учитываться при разработке параллельных программ:

- моменты выполнения командных последовательностей разных потоков могут чередоваться по времени;
- между моментами исполнения команд разных потоков могут выполняться различные временные соотношения (отношения следования); характер этих соотношений зависит от количества и быстродействия процессоров и загрузки вычислительной системы и, тем самым, не может быть определен заранее;
- временные соотношения между моментами исполнения команд могут различаться при разных запусках программ на выполнение, т.е. одной и той же программе при одинаковых исходных данных могут соответствовать разные командные последовательности вследствие разных вариантов чередования моментов работы разных потоков;
- доказательство правильности получаемых результатов должно проводиться для любых возможных временных соотношений для элементов временных траекторий потоков;
- для исключения зависимости результатов выполнения программы от порядка чередования команд разных потоков необходим анализ ситуаций взаимовлияния потоков и разработка методов для их исключения.

Перечисленные моменты свидетельствуют о *существенном повышении сложности параллельного программирования* по сравнению с разработкой "традиционных" последовательных программ.

В завершение следует отметить, что, в самом общем случае, параллельная программа может представлять собой набор процессов, каждый из которых может состоять из нескольких потоков.

## **4.2. Взаимодействие и взаимоисключение потоков**

Одной из причин зависимости результатов выполнения программ от порядка чередования команд может быть разделение одних и тех же данных между одновременно исполняемыми потоками (например, как это осуществляется в выше рассмотренном примере). Данная ситуация может рассматриваться как проявление общей *проблемы использования разделяемых ресурсов* (общих данных, файлов, устройств и т.п.). Для организации разделения ресурсов между несколькими потоками необходимо иметь возможность:

- определения доступности запрашиваемых ресурсов (ресурс свободен и может быть выделен для использования, ресурс уже занят одним из потоков программы и не может использоваться дополнительно каким-либо другим потоком);
- выделения свободного ресурса одному из потоков, запросивших ресурс для использования;
- приостановки (*блокировки*) потоков, выдавших запросы на ресурсы, занятые другими потоками.

Главное требование к механизмам разделения ресурсов является гарантированное обеспечение *использования каждого разделяемого ресурса только одним потоком* от момента выделения ресурса этому потоку до момента освобождения ресурса. Данное требование в литературе обычно именуется *взаимоисключением потоков* (*mutual exclusion*); командные последовательности потоков, в ходе которых поток использует ресурс на условиях взаимоисключения, называется *критической секцией* потока. С использованием последнего понятия условие взаимоисключения потоков может быть сформулировано как требование *нахождения в критических секциях по использованию одного и того же разделяемого ресурса не более чем одного потока*.

Требование взаимоисключения не является единственным к способам организации критических секций; дополнительный перечень необходимых свойств состоит в следующем:

- *Отсутствие взаимной блокировки*. Потоки (по отдельности или совместно) не могут мешать каким-либо потокам обращаться к выполнению своих критических секций;
- *Эффективность*. При наличии нескольких потоков, пытающихся начать выполнение своих критических секций, выбор единственного потока для продолжения осуществляется за конечное (малое) время;
- *Отсутствие бесконечного ожидания*. Поток, пытающийся начать выполнение своей критической секции, гарантированно должен когда-либо получить такую возможность.

При разработке способов обеспечения критических секций обычно предполагается также, что относительные скорости выполнения потоков неизвестны и произвольны, а длительность нахождения потоков в своих критических секциях является конечной.

### **4.2.1. Разработка алгоритма взаимоисключения**

Рассмотрим несколько вариантов программного решения проблемы взаимоисключения (для записи программ используется псевдокод, близкий к языку программирования C++). В каждом из вариантов будет предлагаться некоторый частный способ взаимоисключения потоков с целью демонстрации всех возможных ситуаций при использовании общих разделяемых ресурсов. Последовательное усовершенствование механизма взаимоисключения при рассмотрении вариантов приведет к изложению *алгоритма Деккера*, обеспечивающего взаимоисключение для двух параллельных потоков. Обсуждение способов взаимоисключения будет продолжено далее в пп. 4.2.2 - 4.2.3 рассмотрением *концепции семафоров и мониторов*, которые могут быть

использованы для общего решения проблемы взаимоисключения любого количества взаимодействующих потоков.

### ***Вариант 1 – Жесткая синхронизация***

В первом варианте для взаимоисключения используется управляющая переменная для задания номера потока, имеющего право на использование общего разделяемого ресурса.

```
int ThreadNum=1; // номер потока для доступа к ресурсу
Thread_1() {
    while (1) {
        // повторять, пока право доступа к ресурсу у потока 2
        while ( ThreadNum == 2 );
        <Использование общего ресурса>
        // передача права доступа к ресурсу потоку 2
        ThreadNum = 2;
    }
}
Thread_2() {
    while (1) {
        // повторять, пока право доступа к ресурсу у потока 1
        while ( ThreadNum == 1 );
        < Использование общего ресурса >
        // передача права доступа к ресурсу потоку 1
        ThreadNum = 1;
    }
}
```

Реализованный в программе способ гарантирует взаимоисключение, однако такому решению присущи два существенных недостатка:

- ресурс используется потоками строго последовательно (по очереди) и, как результат, при разном темпе развития потоков общая скорость выполнения программы будет определяться наиболее медленным потоком;
- при завершении работы какого-либо потока другой поток не сможет воспользоваться ресурсом и может оказаться в постоянно блокированном состоянии.

Решение проблемы взаимоисключения подобным образом известно в литературе как способ *жесткой синхронизации*.

### ***Вариант 2 – Потеря взаимоисключения***

В данном варианте для ухода от жесткой синхронизации используются две управляющие переменные, фиксирующие использование потоками разделяемого ресурса.

```
int ResourceThread1 = 0; // =1 - ресурс занят потоком 1
int ResourceThread2 = 0; // =1 - ресурс занят потоком 2
Thread_1() {
    while (1) {
        // повторять, пока ресурс используется потоком 2
        while ( ResourceThread2 == 1 );
        ResourceThread1 = 1;
        < Использование общего ресурса >
        ResourceThread1 = 0;
    }
}
Thread_2() {
    while (1) {
        // повторять, пока ресурс используется потоком 1
        while ( ResourceThread1 == 1 );
```

```

    ResourceThread2 = 1;
    < Использование общего ресурса >
    ResourceThread2 = 0;
}
}

```

Предложенный способ разделения ресурсов устраниет недостатки жесткой синхронизации, однако при этом *теряется гарантия взаимоисключения* – оба потока могут оказаться одновременно в своих критических секциях (это может произойти, например, при переключении между потоками в момент завершения проверки занятости ресурса). Данная проблема возникает вследствие различия моментов проверки и фиксации занятости ресурса.

Следует отметить, что в отдельных случаях взаимоисключение потоков в данном примере может произойти и корректно – все определяется конкретными моментами переключения потоков. Отсюда следует два важных вывода:

- успешность однократного выполнения не может служить доказательством правильности функционирования параллельной программы даже при неизменных параметрах решаемой задачи;
- для выявления ошибочных ситуаций необходима проверка разных временных траекторий выполнения параллельных потоков.

### ***Вариант 3 – Возможность взаимоблокировки***

Возможная попытка в восстановлении взаимоисключения может состоять в установке значений управляющих переменных перед циклом проверки занятости ресурса.

```

int ResourceThread1 = 0; // =1 – ресурс занят потоком 1
int ResourceThread2 = 0; // =1 – ресурс занят потоком 2

Thread_1() {
    while (1) {
        // установить, что поток 1 пытается занять ресурс
        ResourceThread1 = 1;
        // повторять, пока ресурс занят потоком 2
        while ( ResourceThread2 == 1 );
        < Использование общего ресурса >
        ResourceThread1 = 0;
    }
}

Thread_2() {
    while (1) {
        // установить, что поток 2 пытается занять ресурс
        ResourceThread2 = 1;
        // повторять, пока ресурс используется потоком 1
        while ( ResourceThread1 == 1 );
        < Использование общего ресурса >
        ResourceThread2 = 0;
    }
}

```

Представленный вариант восстанавливает взаимоисключение, однако при этом возникает новая проблема – оба потока могут оказаться заблокированными вследствие бесконечного повторения циклов ожидания освобождения ресурсов (что происходит при одновременной установке управляющих переменных в состояние "занято"). Данная проблема известна под названием ситуации *тутика (дедлока или смертельного объятия)* и исключение туников является одной из наиболее важных задач в теории и практике параллельных вычислений. Более подробное рассмотрение темы будет выполнено далее в пп. 4.4; дополнительная информация по проблеме может быть получена в [33].

#### **Вариант 4 – Бесконечное откладывание**

В данном предлагаемом подходе для устранения тупика организуется временное снятие значения занятости управляемых переменных потоков в цикле ожидания ресурса.

```
int ResourceThread1 = 0; // =1 - ресурс занят потоком 1
int ResourceThread2 = 0; // =1 - ресурс занят потоком 2

Thread_1() {
    while (1) {
        ResourceThread1=1; // поток 1 пытается занять ресурс
        // повторять, пока ресурс занят потоком 2
        while ( ResourceThread2 == 1 ) {
            ResourceThread1 = 0; // снятие занятости ресурса
            <временная задержка>
            ResourceThread1 = 1;
        }
        < Использование общего ресурса >
        ResourceThread1 = 0;
    }
}

Thread_2() {
    while (1) {
        ResourceThread2=1; // поток 2 пытается занять ресурс
        // повторять, пока ресурс используется потоком 1
        while ( ResourceThread1 == 1 ) {
            ResourceThread2 = 0; // снятие занятости ресурса
            <временная задержка>
            ResourceThread2 = 1;
        }
        < Использование общего ресурса >
        ResourceThread2 = 0;
    }
}
```

Длительность временной задержки в циклах ожидания должна определяться при помощи некоторого случайного датчика. При таких условиях реализованный алгоритм обеспечивает взаимоисключение и исключает возникновение тупиков, но опять таки не лишен существенного недостатка (перед чтением следующего текста попытайтесь определить этот недостаток). Проблема состоит в том, что потенциально решение вопроса о выделении может откладываться до бесконечности (при синхронном выполнении потоков). Данная ситуация известна под наименованием *бесконечное откладывание (starvation)* или *длительной блокировкой (live lock)*.

#### **Алгоритм Деккера**

В алгоритме, впервые предложенным Деккером [31]. предлагается объединение предложений вариантов 1 и 4 решения проблемы взаимоисключения.

```
int ThreadNum=1; // номер потока для доступа к ресурсу
int ResourceThread1 = 0; // =1 - ресурс занят потоком 1
int ResourceThread2 = 0; // =1 - ресурс занят потоком 2

Thread_1() {
    while (1) {
        ResourceThread1=1; // поток 1 пытается занять ресурс
        // цикл ожидания доступа к ресурсу
        while ( ResourceThread2 == 1 ) {
            if ( ThreadNum == 2 ) {
                ResourceThread1 = 0;
                // повторять, пока ресурс занят потоком 2
                while ( ThreadNum == 2 );
                ResourceThread1 = 1;
            }
        }
    }
}
```

```

    < Использование общего ресурса >
    ThreadNum      = 2;
    ResourceThread1 = 0;
}
}

Thread_2() {
while (1) {
    ResourceThread2=1; // поток 2 пытается занять ресурс
    // цикл ожидания доступа к ресурсу
    while ( ResourceThread1 == 1 ) {
        if ( ThreadNum == 1 ) {
            ResourceThread2 = 0;
            // повторять, пока ресурс используется потоком 1
            while ( ThreadNum == 1 );
            ResourceThread2 = 1;
        }
    }
    < Использование общего ресурса >
    ThreadNum      = 1;
    ResourceThread2 = 0;
}
}

```

Алгоритм Деккера гарантирует корректное решение проблемы взаимоисключения для двух потоков. Управляющие переменные *ResourceThread1*, *ResourceThread1* обеспечивают взаимоисключение, переменная *ThreadNum* исключает возможность бесконечного откладывания. Если оба потока пытаются получить доступ к ресурсу, то поток, номер которого указан в *ThreadNum*, продолжает проверку возможности доступа к ресурсу (внешний цикл ожидания ресурса). Другой же поток в этом случае снимает свой запрос на ресурс, ожидает своей очереди доступа к ресурсу (внутренний цикл ожидания) и возобновляет свой запрос на ресурс.

Алгоритм Деккера может быть обобщен на случай произвольного количества потоков, однако такое обобщение приводит к заметному усложнению выполняемых действий. Кроме того, программное решение проблемы взаимоисключения потоков приводит к нерациональному использованию процессорного времени ЭВМ (потоку, ожидающему освобождения ресурса, постоянно требуется процессор для проверки возможности продолжения – *активное ожидание* (*busy wait*)).

#### 4.2.2. Семафоры

Приведенные примеры показывают, что организация критических секций обычными средствами требует определенных усилий. Как результат, для решения вопросов взаимоисключения может оказаться целесообразной разработка новых (специальных) механизмов. Один из классических подходов в этом ряду – семафоры, предложенные Дейкстрой еще в середине 1960-х годов.

Под *семафором S* понимается [5] переменная особого типа, значение которой может опрашиваться и изменяться только при помощи специальных операций *P(S)* и *V(S)*, реализуемых в соответствии со следующими алгоритмами:

- операция P(S)

если  $S > 0$

то  $S = S - 1$

иначе < ожидать S >

- операция V(S)

если < один или несколько потоков ожидают S >

то < снять ожидание у одного из ожидающих потоков >

иначе  $S = S + 1$

Принципиальным в понимании семафоров является то, что операции  $P(S)$  и  $V(S)$  предполагаются неделимыми (атомарными), что гарантирует взаимоисключение при использование общих семафоров (для обеспечения неделимости операции обслуживания семафоров обычно реализуются средствами операционной системы).

Различают два основных типа семафоров. Двоичные семафоры принимают только значения 0 и 1, область значений общих семафоров – неотрицательные целые значения. В момент создания семафоры инициализируются некоторым целым значением.

Семафоры широко используются для синхронизации и взаимоисключения потоков. Так, например, проблема взаимоисключения при помощи семафоров может иметь следующее простое решение.

```
Semaphore Sem=1; // семафор взаимоисключения потоков
Thread_1() {
    while (1) {
        // проверить семафор и ждать, если ресурс занят
        P(Sem);
        < Использование общего ресурса >
        // освободить один из ожидающих ресурса потоков
        // увеличить семафор, если нет ожидающих потоков
        V(Sem);
    }
}
Thread_2() {
    while (1) {
        // проверить семафор и ждать, если ресурс занят
        P(Sem);
        < Использование общего ресурса >
        // освободить один из ожидающих ресурса потоков
        // увеличить семафор, если нет ожидающих потоков
        V(Sem);
    }
}
```

Приведенный пример рассматривает взаимоисключение только двух потоков, но, как можно заметить, совершенно аналогично может быть организовано взаимоисключение произвольного количества потоков.

Завершая рассмотрение данной темы, отметим, что на практике в разных средах выполнения наряду с поддержкой "стандартных" семафоров реализуются некоторые их разновидности – мьютексы (*mutex*), замки (*lock*) и др. Вариации касаются допустимого набора значений для переменных семафоров, рекурсивности вызова, введения дополнительного набора операций и т.д.

#### 4.2.3. Мониторы

Несмотря на то, что семафоры в значительной степени упрощают проблему организации критических секций, тем не менее, семафоры являются достаточно низкоуровневым средством синхронизации потоков. Взаимосвязь семафоров и критических секций обеспечивается только на логическом уровне. Нерегламентированное использование семафоров приводит к усложнению схемы параллельного выполнения разрабатываемой программы. Более высокоуровневым механизмом синхронизации являются мониторы.

Мониторы представляют собой программные модули (объекты), которые реализуют (инкапсулируют) все необходимые действия с разделяемым ресурсом (см., например, [5]). Общий формат определения монитора может быть представлен следующим образом:

```
Monitor <Name> {
    <объявления переменных>
    <операторы инициализации>
    <процедуры монитора>
}
```

Как можно заметить, описание монитора достаточно близко совпадает с описанием класса в алгоритмическом языке C++. Принципиальное отличие состоит в том, что процедуры монитора, в обязательном порядке, выполняются *в режиме взаимоисключения*, т.е. при выполнении какой-либо процедуры монитора все остальные попытки вызова других процедур этого же монитора блокируются. Обеспечение такого правила выполнения процедур монитора должно осуществляться средой выполнения, в которой поддерживается концепция мониторов.

Как пример использования монитора можно рассмотреть задачу организации доступа к общей переменной – возможный вариант монитора для этой цели может быть реализован в виде:

```
Monitor SharedMem {
    int N=0; // Общая переменная
    procedure Set (int v) { // Операция записи
        N = v;
    }
    procedure Get (int &v) { // Операция чтения
        v = N;
    }
    procedure Inc () { // Операция изменения
        N = N + 1;
    }
}
```

Помимо взаимоисключения процедур другим важным свойством понятия монитора является его полная изолированность от остального кода программы:

- Переменные монитора недоступны вне монитора и могут обрабатываться только процедурами монитора;
- Вне монитора доступны только процедуры монитора;
- Переменные, объявленные вне монитора, недоступны внутри монитора.

Подобная локализация (инкапсуляция) и объединение в рамках монитора всех критических секций потоков приводит к значительному снижению сложности логики параллельного выполнения.

В ряде случаев для процедур монитора может потребоваться приостановка (блокировка) до выполнения каких-либо условий – например, при реализации семафоров при помощи монитора процедура занятия семафора должна блокироваться, если данный семафор уже является занятым. Для решения таких проблем в мониторах вводится дополнительный механизм условных переменных.

*Условные переменные* – это объекты специального типа *cond*, используемые для организации приостановки работы процедур монитора до выполнения определенных логических условий. Действия с условными переменными осуществляются при помощи двух основных операций:

- **wait(cv)** – операция ожидания наступления события; при этом для процедуры, выполнившую данную операцию, снимается блокировка и процедуры монитора снова

становятся доступными для использования (несмотря на то, что приостановленные при помощи операции *wait* процедуры еще не завершили свое выполнение);

– **signal(cv)** – операция для объявления наступления события; в результате выполнения данной операции одна из процедур, ожидающих данного события, становится готовой для продолжения (если ожидающих данного события процедур нет, сигнал о наступлении события теряется)

(отметим, что взаимосвязь условных переменных и событий, им соответствующих, устанавливается только на логическом уровне).

С использованием механизма условных переменных можно провести, например, реализацию семафором при помощи монитора:

```
Monitor Semaphore {  
    int s = 1; // Счетчик семафора  
    cond cv; // Переменная для события s>0  
    procedure Psem () { // Занятие семафора  
        while ( s == 0 ) wait(cv);  
        s = s - 1;  
    }  
    procedure Vsem () { // Освобождение семафора  
        s = s + 1;  
        signal(cv)  
    }  
}
```

(следует заметить, что и, обратно, монитор может быть реализован с использованием семафоров).

Понятие условных переменных достаточно близко понятию семафора, однако есть и существенные различия. Операция *wait* всегда приостанавливает поток, а операция *P* семафора блокирует поток только в случае, если переменная семафора равна нулю. В свою очередь, операция *signal* не выполняет никаких действий, если нет ожидающих потоков, в то время как операция *V* семафора либо активизирует один из блокированных поток либо увеличивает значение переменной семафора.

Различают две различные схемы продолжения работы монитора после выполнения операции *signal*:

– *синхронный* способ, при котором процедура монитора, выполнившая операцию *signal*, приостанавливается, а для продолжения работы выбирается одна из процедур монитора, ожидающих данного события – в литературе мониторы такого типа обычно называют *мониторами Хоара*,

– *асинхронный* способ, для которого порядок действий является обратным по сравнению с мониторами Хоара: одна из процедур монитора, ожидающих данного события, переводится в состояние готовности для выполнения, и далее продолжается работа процедуры монитора, выполнившая операцию *signal* – мониторы такого типа обычно называют *мониторами Меса*.

Наибольшее распространение получила вторая схема – именно такой подход используется в операционной системе Unix, языке программирования Java и библиотеке Pthreads.

Подводя итог, можно отметить, что мониторы являются предпочтительным способом организации взаимоисключения. При этом, если в среде выполнения параллельных программ мониторы в явном виде не поддерживаются, использование мониторов можно промоделировать при помощи семафоров.

## 4.3. Синхронизация потоков

Рассмотренная в предыдущем подразделе задача проблема взаимоисключения, на самом деле, является частным случаем проблемы синхронизации параллельно выполняемых потоков. Необходимость синхронизации обуславливается обстоятельством, что не все возможные траектории совместно выполняемых потоков являются допустимыми (так, например, при использовании общих ресурсов требуется обеспечить взаимоисключение). В самом общем виде, *синхронизация* может быть обеспечена при помощи задания необходимых логических условий, которые должны выполняться в соответствующих точках траекторий потоков.

Организация синхронизации является важной частью разработки параллельной программы. Принципиальный момент состоит в определении полного набора синхронизирующих действий, гарантирующих гарантированное исключение недопустимых траекторий параллельной программы – в этом случае говорят, что программа обладает *свойством безопасности*. Строгость данного свойства может быть несколько снижена – так, набор синхронизирующих действий можно ограничить требованием обеспечения достижимости допустимых траекторий программы – такое поведение обычно именуется *свойством живучести*.

В числе наиболее широко используемых общих механизмов синхронизации (помимо средств взаимоисключения) – *условные переменные* и *барьерная синхронизация*.

### 4.3.1. Условные переменные

Условные переменные, используемые для синхронизации потоков, практически совпадают с аналогичными средствами мониторов (см. п. 4.2.3). Т. е., как и ранее, условные переменные – это объекты некоторого специального типа, используемые для организации синхронизации потоков. Основные операции с условными переменными:

- **wait(cv)** – ожидание события, связанного с условной переменной;
- **signal(cv)** – объявление наступления события.

При выполнении операции *wait* поток блокируется; в результате выполнения операции *signal* один из потоков, ожидающих данного события, переводится в состояние готовности для выполнения (как и ранее, при отсутствии ожидающих потоков сигнал о наступлении события теряется).

При реализации механизма условных переменных в разных средах выполнения рассмотренный выше набор операций обычно расширяется. В числе подобных расширений:

- **empty(cv)** – проверка наличия потоков, ожидающих события;
- **broadcast(cv)** – объявление наступления события для всех ожидающих потоков (все ожидающие данного события потоки переводятся в состояние готовности для выполнения).

### 4.3.2. Барьерная синхронизация

Барьерная синхронизация является частным вариантом организации согласованного выполнения потоков и состоит в выделении точек в траекториях параллельно выполняемых потоков, таких, что выполнение потоков может быть продолжено только в случае, когда все потоки достигнут своих точек барьерной синхронизации. В качестве примера необходимости такой синхронизации можно привести параллельный алгоритм, итерации которого выполняются отдельными потоками и перед переходом к каждой следующей итерации все потоки должны завершить свои текущие итерации.

Операция барьерной синхронизации обычно именуется как *barrier()*. Важно отметить, что данная операция является коллективной и должна быть выполнена в каждом потоке,

участвующем в барьерной синхронизации. Как правило, в барьерной синхронизации принимают участие все параллельно выполняемые потоки программы, хотя в различных средах выполнения может быть предусмотрена возможность барьерной синхронизации и для отдельных групп выполняемых потоков.

#### 4.4. Взаимоблокировка потоков

В самом общем виде *взаимоблокировка* может быть определена [33] как ситуация, в которой один или несколько потоков ожидают какого-либо события, которое никогда не произойдет (в научно-технической литературе такая ситуация чаще всего называется как *тупик*, а также *дедлок* или *смертельное объятие*). Важно отметить, что состояние тупика может наступить не только вследствие логических ошибок, допущенных при разработке параллельных программ, но и в результате возникновения тех или иных событий в вычислительной системе (выход из строя отдельных устройств, нехватка ресурсов и т.п.). Простой пример тупика может состоять в следующем. Пусть имеется два потока, каждый из которых в монопольном режиме обрабатывает собственный файл данных. Ситуация тупика возникнет, например, если первому потоку для продолжения работы потребуются файл второго потока и одновременно второму потоку окажется необходимым файл первого потока (см. рис. 4.3).

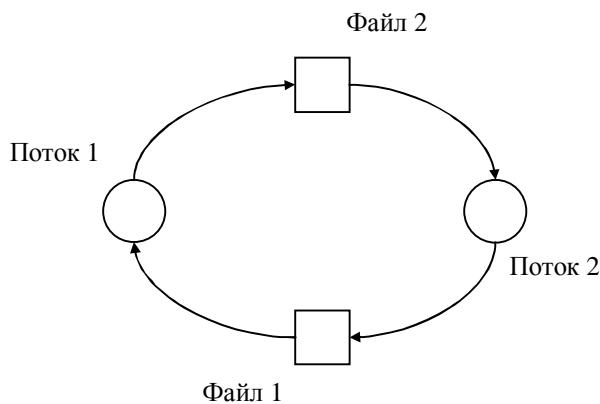


Рис. 4.3. Пример ситуации тупика

Проблема тупиков имеет многоплановый характер. Это и сложность диагностирования состояния тупика (система выполняет длительные расчеты или "зависла" из-за тупика), и необходимость определенных специальных действий для выхода из тупика, и возможность потери данных при восстановлении системы при устранении тупика.

В данном разделе будет рассмотрен один из аспектов проблемы тупика – анализ причин возникновения тупиковых ситуаций при использовании разделяемых ресурсов и разработка на этой основе методов предотвращения тупиков. Дополнительная информация по теме может быть получена в [33].

Могут быть выделены следующие **необходимые условия тупика** [33]:

- потоки требуют предоставления им права монопольного управления ресурсами, которые им выделяются (*условие взаимоисключения*);
- потоки удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (*условие ожидания ресурсов*);
- ресурсы нельзя отобрать у потоков, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (*условие неперераспределемости*);

- существует кольцевая цепь потоков, в которой каждый поток удерживает за собой один или более ресурсов, требующихся следующему потоку цепи (условие *кругового ожидания*).

Как результат, для обеспечения отсутствия тупиков необходимо исключить возникновение, по крайней мере, одного из рассмотренных условий. Далее будет предложена модель программы в виде графа "поток-ресурс", позволяющего обнаруживать ситуации кругового ожидания [36].

#### 4.4.1. Модель программы в виде графа "поток-ресурс"

*Состояние программы* может быть представлено в виде ориентированного графа  $(V, E)$  со следующей интерпретацией и условиями:

1. Множество  $V$  разделено на два взаимно пересекающихся подмножества  $P$  и  $R$ , представляющие *потоки*

$$P = (p_1, p_2, \dots, p_n)$$

и *ресурсы*

$$R = (R_1, R_2, \dots, R_m)$$

программы.

2. Граф является "двудольным" по отношению к подмножествам вершин  $P$  и  $R$ , т.е. каждое ребро  $e \in E$  соединяет вершину  $P$  с вершиной  $R$ . Если ребро  $e$  имеет вид  $e = (p_i, R_j)$ , то  $e$  есть ребро *запроса* и интерпретируется как запрос от потока  $p_i$  на единицу ресурса  $R_j$ . Если ребро  $e$  имеет вид  $e = (R_j, p_i)$ , то  $e$  есть ребро *назначения* и выражает назначение единицы ресурса  $R_j$  потоку  $p_i$ .

3. Для каждого ресурса  $R_j \in R$  существует целое  $k_j \geq 0$ , обозначающее количество единиц ресурса  $R_j$ .

4. Пусть  $|(a, b)|$  - число ребер, направленных от вершины  $a$  к вершине  $b$ . Тогда при принятых обозначениях для ребер графа должны выполняться условия:

- Может быть сделано не более  $k_j$  назначений (распределений) для ресурса  $R_j$ , т.е.

$$\sum_i |(R_j, p_i)| \leq k_j, \quad 1 \leq j \leq m;$$

- Сумма запросов и распределений относительно любого потока для конкретного ресурса не может превышать количества доступных единиц, т.е.

$$|(R_j, p_i)| + |(p_i, R_j)| \leq k_j, \quad 1 \leq i \leq n, \quad 1 \leq j \leq m.$$

Граф, построенный с соблюдением всех перечисленных правил, именуется в литературе как *граф "поток-ресурс"*. Для примера, на рис. 4.3 приведен граф программы, в которой ресурс 1 (файл 1) выделен потоку 1, который, в свою очередь, выдал запрос на ресурс 2 (файл 2). Поток 2 владеет ресурсом 2 и нуждается для своего продолжения в ресурсе 1.

Состояние программы, представленное в виде графа "поток-ресурс", изменяется только в результате *запросов, освобождений* или *приобретений* ресурсов каким-либо из потоков программы.

**Запрос.** Если программа находится в состоянии  $S$  и поток  $p_i$  не имеет невыполненных запросов, то  $p_i$  может запросить любое число ресурсов (в пределах ограничения 4). Тогда программа переходит в состояние  $T$

$$S \xrightarrow{i} T.$$

Состояние  $T$  отличается от  $S$  только дополнительными ребрами запроса от  $p_i$  к затребованным ресурсам.

**Приобретение.** Операционная система может изменить состояние программы  $S$  на состояние  $T$  в результате операции приобретения ресурсов потоком  $p_i$  тогда и только тогда, когда  $p_i$  имеет запросы на выделение ресурсов и все такие запросы могут быть удовлетворены, т.е. если

$$\forall R_j : (p_i, R_j) \in E \Rightarrow (p_i, R_j) + \sum_l |(R_j, p_l)| \leq k_j .$$

Граф  $T$  идентичен  $S$  за исключением того, что все ребра запроса  $(p_i, R_j)$  для  $p_i$  обратны ребрам  $(R_j, p_i)$ , что отражает выполненное распределение ресурсов.

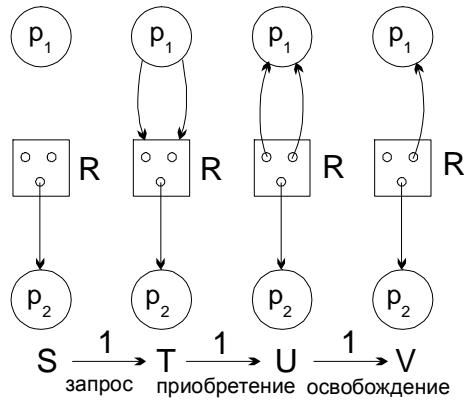


Рис. 4.4. Пример переходов программы из состояния в состояние

**Освобождение.** Поток  $p_i$  может вызвать переход из состояния  $S$  в состояние  $T$  с помощью освобождения ресурсов тогда и только тогда, когда  $p_i$  не имеет запросов, а имеет некоторые распределенные ресурсы, т.е.

$$\forall R_j : (p_i, R_j) \notin E, \exists R_j : (R_j, p_i) \in E .$$

В этой операции  $p_i$  может освободить любое непустое подмножество своих ресурсов. Результирующее состояние  $T$  идентично исходному состоянию  $S$  за исключением того, что в  $T$  отсутствуют некоторые ребра приобретения из  $S$  (из  $S$  удаляются ребра  $(R_j, p_i)$  каждой освобожденной единицы ресурса  $R_j$ ).

Для примера на рис. 4.4. показаны состояния программы с одним ресурсом емкости 3 и двумя потоками после выполнения операций запроса, приобретения и освобождения ресурсов для первого потока.

При рассмотрении переходов программы из состояния в состояние важно отметить, что поведение потоков является недетерминированным – при соблюдении приведенных выше ограничений выполнение любой операции любого потока возможно в любое время.

#### 4.4.2. Описание возможных изменений состояния программы

Определение состояния программы и операций перехода между состояниями позволяет сформировать модель параллельной программы следующего вида.

Под *программой* будем понимать систему

$$\langle \Sigma, P \rangle ,$$

где  $\Sigma$  есть множество состояний программы ( $S, T, U, \dots$ ), а  $P$  представляет множество потоков  $(p_1, p_2, L, p_n)$ . Поток  $p_i \in P$  есть частичная функция, отображающая состояния программы в непустые подмножества состояний

$$p_i : \Sigma \rightarrow \{\Sigma\},$$

где  $\{\Sigma\}$  есть множество всех подмножеств  $\Sigma$ . Обозначим множество состояний, в которые может перейти программа при помощи потока  $p_i$  (*область значений потока  $p_i$* ) при нахождении программы в состоянии  $S$  через  $p_i(S)$ . Возможность перехода программы из состояния  $S$  в состояние  $T$  в результате некоторой операции над ресурсами в потоке  $p_i$  (т.е.  $T \in p_i(S)$ ) будем пояснить при помощи записи

$$S \xrightarrow{i} T.$$

Обобщим данное обозначение для указания достижимости состояния  $T$  из состояния  $S$  в результате выполнения некоторого произвольного количества переходов в программе

$$\begin{aligned} S \xrightarrow{*} T \Leftrightarrow & (S = T) \vee (\exists p_i \in P : S \xrightarrow{i} T) \vee \\ & (\exists p_i \in P, U \in \Sigma : S \xrightarrow{i} U, U \xrightarrow{*} T) \end{aligned}$$

#### 4.4.3. Обнаружение и исключение тупиков

С учетом построенной модели и введенных обозначений можно выделить ряд ситуаций, возникающих при выполнении программы и представляющих интерес при рассмотрении проблемы тупика:

- поток  $p_i$  заблокирован в состоянии  $S$ , если программа не может изменить свое состояние при помощи этого потока, т.е. если  $p_i(S) = \emptyset$ ;
- поток  $p_i$  находится в *тупике* в состоянии  $S$ , если этот поток является заблокированным в любом состоянии  $T$ , достижимом из состояния  $S$ , т.е.

$$\forall T : S \xrightarrow{*} T \Rightarrow p_i(T) = \emptyset;$$

- *состояние  $S$*  называется *тупиковым*, если существует поток  $p_i$ , находящийся в тупике в этом состоянии;
- *состояние  $S$*  есть *безопасное состояние*, если любое состояние  $T$ , достижимое из  $S$ , не является тупиковым.

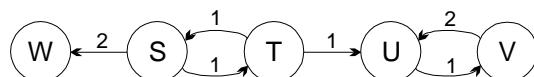


Рис. 4.5. Пример графа переходов программы

Для примера на рис. 4.5 приведен график переходов программы, в котором состояния  $U$  и  $V$  являются безопасными, состояния  $S$  и  $T$  и  $W$  не являются безопасными, а состояние  $W$  есть состояние тупика.

Рассмотренная модель программы может быть использована для определения возможных состояний программы, обнаружения и недопущения тупиков. В качестве возможных теоретических результатов такого анализа может быть приведена теорема [36].

**Теорема.** Граф "поток-ресурс" для состояния программы с ресурсами единичной емкости указывает на состояние тупика тогда и только тогда, когда он содержит цикл.

Дополнительный материал по исследованию данной модели может быть получен в [36].

#### 4.5. Классические задачи синхронизации

В ходе изучения проблем параллельного программирования и разработки методов их решения сформировался набор некоторых "классических" задач, на которых принято

демонстрировать результаты применения новых разрабатываемых подходов. В числе этих задач:

- Задача "Производители-Потребители" (*Producer-Consumer problem*);
- Задача "Читатели-Писатели" (*Readers-Writers problem*);
- Задача "Обедающие философы" (*Dining Philosopher problem*);
- Задача "Спящий брадобрей" (*Sleeping Barber problem*).

Далее будет представлено описание этих задач и рассмотрены примеры их возможных решений.

#### 4.5.1. Задача "Производители-Потребители"

В научно-технической литературе существует достаточно большое количество вариантов постановки данной задачи. В наиболее простом случае предполагается, что существует два потока, один из которых (*производитель*) генерирует сообщения (*изделия*), а второй поток (*потребитель*) их принимает для последующей обработки. Потоки взаимодействуют через некоторую область памяти (*хранилище*), в которой производитель размещает свои генерируемые сообщения и из которой эти сообщения извлекаются потребителем (см. рис. 4.5).

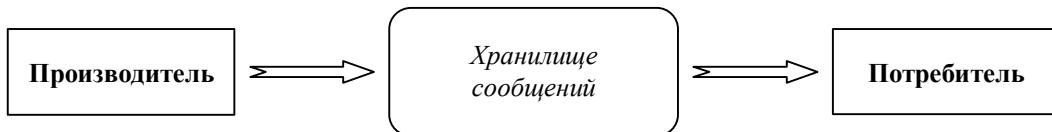


Рис. 4.5. Общая схема задачи "Производители-Потребители"

Рассмотрев постановку данной задачи, можно заметить, что хранилище сообщений представляет собой не что иное, как общий разделяемый ресурс и использование этого ресурса должно быть построено по правилам взаимоисключения. Кроме того, следует учитывать, что потребление ресурса иногда может оказаться невозможным (отсутствие сообщений в хранилище), а при добавлении сообщений в хранилище могут происходить задержки (в случае полного заполнения хранилища).

Далее будет представлено решение этой задачи с использованием семафоров, однако перед ознакомлением с этим решением полезно попытаться разработать решение самостоятельно. Дополнительно можно порекомендовать ознакомиться и с другими возможными решениями данной задачи, используя для этого, например, работу [33]. Кроме того, крайне полезно не ограничиваться только "теоретическим" знакомством с этими алгоритмами, а попытаться их реализовать и проверить в той или иной среде выполнения параллельных программ (можно заметить, что все приведенные замечания будут справедливы и при рассмотрении всех последующих задач).

Для организации работы используем три семафора:

- Access – двоичный семафор для организации взаимоисключения при доступе к хранилищу;
- Full – общий семафор, блокирующий поток-производитель при попытке записи сообщения в полностью заполненное хранилище (в переменной семафора будет хранится количество имеющихся свободных мест для сообщений в хранилище);
- Empty – общий семафор, блокирующий поток-потребитель при попытке чтения сообщения из пустого хранилище (в переменной семафора будет хранится количество имеющихся сообщений в хранилище).

Возможная реализация взаимоисключения потоков может состоять в следующем.

```
Semaphore Access = 1; // Семафор взаимоисключения доступа  
Semaphore Full    = n; // Семафор блокировки записи в полное хранилище
```

```

Semaphore Empty = 0; // Семафор блокировки чтения из пустого хранилища
Producer(){
    <Генерация нового сообщения>
    P(Full); // Доступ только при наличии пустых мест в хранилище
    P(Access); // Блокировка доступа к хранилищу
    <Запись сообщения в хранилище>
    V(Access); // Снятие блокировки доступа к хранилишу
    V(Empty); // Отметка наличия сообщений в хранилище
}
Consumer(){
    P(Empty); // Доступ только при наличии сообщений в хранилище
    P(Access); // Блокировка доступа к хранилищу
    <Чтение сообщения из хранилища>
    V(Access); // Снятие блокировки доступа к хранилишу
    V(Full); // Отметка наличия пустых мест в хранилище
    <Обработка полученного сообщения>
}

```

В качестве самостоятельного упражнения может быть предпринята попытка разработка решения задачи при помощи монитора и/или рассмотрение более усложненных постановок задачи "Производитель-Потребитель" (произвольное количество потоков, двусторонняя передача сообщений и т.п.).

#### 4.5.2. Задача "Читатели-Писатели"

Возможная простая постановка этой задачи состоит в следующем. Пусть имеется некоторая область памяти – хранилище данных, с которым одновременно работают несколько потоков. По характеру использования хранилища потоки могут быть разделены на два типа. Одна группа – это *потоки-читатели*, которые осуществляют только чтение (без удаления) хранилища. Другая – оставшаяся часть потоков – это *потоки-писатели*, которые выполняют запись новых значений имеющихся в хранилище данных (см. рис. 4.6).

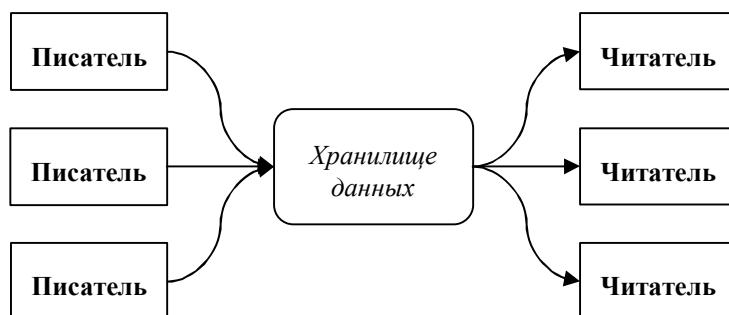


Рис. 4.6. Общая схема задачи "Читатели-Писатели"

При рассмотрении этой задачи предполагается, что потоки-читатели не изменяют каких-либо параметров хранилища и могут, тем самым, работать одновременно не мешая друг другу. Для потоков-писателей ситуация обратная – предполагается, что запись не может выполняться несколькими потоками одновременно и, понятно, во время записи не допускаются какие-либо операции чтения.

Постановка задачи может различаться в правилах разрешения ситуации обращения потока-писателя к хранилищу. Если при попытке записи имеются активные потоки-читатели, то поток-писатель должен быть заблокирован до завершения работы потоков-читателей. Вопрос состоит в том, что делать с новыми поступающими запросами на чтение. Возможный вариант – отдать предпочтение потокам-читателям (т.е. новые потоки-читатели могут начинать свою работу, не обращая внимание на блокированный процесс-писатель), однако, в этом случае, блокировка потока-писателя может

продолжаться бесконечно долго. Альтернативный подход – блокировка новых потоков-читателей, появившихся после блокировки потока-писателя.

Для решения поставленной задачи снова используем семафоры. Введем следующие переменные:

- ReadCount – переменная-счетчик количества активных потоков-читателей;
- ReadSem – двоичный семафор для взаимоисключения доступа к переменной *ReadCount*;
- Access – двоичный семафор для организации взаимоисключения при доступе к хранилищу.

Возможная реализация синхронизации взаимодействия потоков может состоять в следующем.

```
int ReadCount = 0;      // Счетчик количества активных потоков-читателей
Semaphore ReadSem = 1; // Семафор доступа к переменной ReadCount
Semaphore Access = 1; // Семафор доступа к хранилищу

Writer(){ // Поток-писатель
    P(Access);        // Блокировка доступа к хранилищу
    <Выполнение операции записи>
    V(Access);        // Снятие блокировки доступа к хранилищу
}

Reader(){ // Поток-читатель
    P(ReadSem);       // Блокировка доступа к переменной ReadCount
    ReadCount++;      // Изменение счетчика активных потоков-читателей
    if( ReadCount == 1 )
        P(Access); // Блокировка доступа к хранилищу (если поток-читатель первый)
    V(ReadSem);       // Снятие блокировки доступа к переменной ReadCount
    <Выполнение операции чтения>
    P(ReadSem);       // Блокировка доступа к переменной ReadCount
    ReadCount--;      // Изменение счетчика активных потоков-читателей
    if( ReadCount == 0 ) // Снятие блокировки доступа к хранилищу
        V(Access); // (если завершается последний поток-читатель)
    V(ReadSem);       // Снятие блокировки доступа к переменной ReadCount
}
```

В качестве самостоятельного задания предлагается провести анализ приведенного решения и определить, какой вариант поведения новых потоков-читателей при наличии блокированных потоков-писателей в данном алгоритме обеспечивается. Если предлагаемый алгоритм отдает предпочтение потокам-читателям, следует попытаться разработать решение, справедливое по отношению к потокам-писателям.

Дополнительная информация по данной задаче может быть получена, например, в [33].

#### 4.5.3. Задача "Обедающие философы"

Данная задача является одной из наиболее известных в области параллельного программирования. Если задача "Читатели-Писатели" помогает демонстрировать методы параллельного и исключительного доступа к одному общему ресурсу, то задача "Обедающие философы" позволяет рассмотреть способы доступа нескольких потоков к нескольким разделяемым ресурсам.

Исходная формулировка задачи, впервые предложенная Э. Дейкстрой, выглядит следующим образом. Представляется ситуация, в которой пять философов располагаются за круглым столом. При этом философы либо размышляют, либо кушают. Для приема пищи в центре стола большое блюдо с неограниченным количеством спагетти и тарелки, по одной перед каждым философом. Предполагается, что поесть спагетти можно только с использованием двух вилок. Для этого на столе располагается ровно пять вилок – по одной между тарелками философов (см. рис. 4.7).

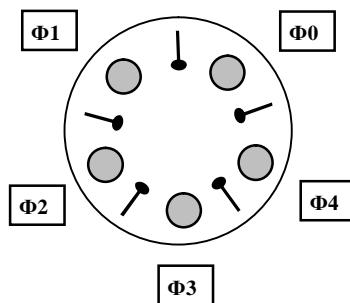


Рис. 4.7. Общая схема задачи "Обедающие философы"

Для того, чтобы приступить к еде, философ должен взять вилки слева и справа (если они не заняты), наложить спагетти из большого блюда в свою тарелку, поесть, а затем обязательно положить вилки на свои места для их повторного использования (проблема чистоты вилок в задаче не рассматривается).

Нетрудно заметить, что в данной задаче философы представляют собой потоки, а вилки – общие разделяемые ресурсы. Тогда первое очевидное, на первый взгляд, решение состоит в том, чтобы для каждой вилки (ресурса) ввести отдельный семафор для блокировки философа (потока) в ситуации, когда нужная для еды вилка уже занята соседним философом. Кроме того, можно применить некоторое регламентирующее правило порядка взятия вилок – например, философ сначала берет левую вилку, затем правую.

Итак, получаемый в результате алгоритм деятельности каждого философа состоит в следующем – как только философ приступает к еде, он пытается взять левую вилку. Если она занята, философ ждет ее освобождения и, в конце концов, ее получает. Затем философ пытается взять правую вилку. И опять же, если вилка занята, философ снова ждет ее освобождения (при этом, левую вилку он по прежнему хранит у себя). После получения правой вилки философ ест спагетти, после чего освобождает обе вилки.

Возможная реализация предложенной схемы (опять же с использованием семафоров) может состоять в следующем.

```
Semaphore fork[5] = { 1, 1, 1, 1, 1 }; // Семафоры доступа к вилкам
Philosopher(){ // Поток -философ (для всех философов одинаковый)
    // i - номер философа
    while (1) {
        P(fork[i]);           // Доступ к левой вилке
        P(fork[(i+1)%5]);   // Доступ к правой вилке
        <Питание>
        V(fork[i]); V(fork[(i+1)%5]) // Освобождение вилок
        <Размышление>
    }
}
```

(выражение  $(i+1)\%5$  определяет номер правой вилки,  $\%$  есть операция получения остатка от целого деления в алгоритмическом языке С).

Внимательно проанализируйте представленный алгоритм. После тщательного изучения можно увидеть, что данное решение может приводить к тупиковым ситуациям – например, когда все философы одновременно проголодаются и каждый из них возьмет свои левые вилки. В результате, правые вилки для всех философов окажутся занятыми и философы перейдут к бесконечному ожиданию (отметим, как сложно выявить подобную ситуацию при помощи тестов; кроме того, подобную ошибочную ситуацию сложно повторить при повторных запусках программы).

Возможны различные варианты исправления рассмотренного алгоритма. Для этого надо устранить одно из условий возникновения тупика (см. начало данного подраздела) – например, попытаться избежать кругового ожидания. Для этого можно изменить порядок

взятия вилок для одного из философов (например, для четвертого), который должен брать сначала правую вилку, а только затем левую. Получаемое в результате решение выглядит следующим образом.

```
Semaphore fork[5] = { 1, 1, 1, 1, 1 }; // Семафоры доступа к вилкам
Prilosopher(){ // Поток-философ (для всех философов, кроме четвертого)
    // i - номер философа
    while (1) {
        P(fork[i]);           // Доступ к левой вилке
        P(fork[(i+1)%5]);   // Доступ к правой вилке
        <Питание>
        V(fork[i]); V(fork[(i+1)%5]) // Освобождение вилок
        <Размышление>
    }
}
Prilosopher4(){ // Поток для четвертого философа)
    while (1) {
        P(fork[0]); // Доступ к правой вилке
        P(fork[4]); // Доступ к левой вилке
        <Питание>
        V(fork[0]); V(fork[4]) // Освобождение вилок
        <Размышление>
    }
}
```

Изучение нового варианта алгоритма синхронизации показывает, что он гарантирует отсутствие тупиков.

В качестве самостоятельного задания можно предложить выполнить некоторую вариацию постановки задачи и разработать свои варианты алгоритмов синхронизации (так, можно предложить правило, по которому философ берет вилки только в том случае, если они обе свободны, и т.п.).

Дополнительная информация по данной задаче может быть получена, например, в [33].

#### 4.5.4. Задача "Спящий парикмахер"

Данная задача также в числе широко используемых примеров для демонстрации проблем синхронизации. На примере этой задачи можно показать методы последовательного доступа к набору разделяемых ресурсов и рассмотреть организацию вычислений в соответствии со схемой "клиент-сервер".

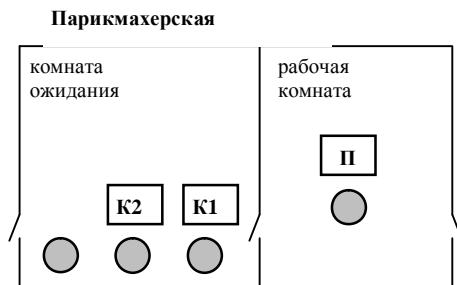


Рис. 4.8. Общая схема задачи "Спящий парикмахер"  
(К – клиенты, П – парикмахер)

Смысловая окраска задачи "Спящий парикмахер" состоит в следующем (см. рис. 4.8). Обсуждается проблема обслуживания клиентов парикмахерской. В парикмахерской имеется два помещения: комната ожидания, в которой ограниченное количество мест, и рабочая комната с единственным креслом, в котором располагается обслуживаемый клиент. Посетители заходят в парикмахерскую – если комната ожидания заполнена, то поворачиваются и уходят; иначе занимают свободные места и засыпают, ожидая своей очереди к парикмахеру. Парикмахер, если есть клиенты, приглашает одного из них в

рабочую комнату, и подстригает его. После стрижки клиент покидает парикмахерскую, а парикмахер приглашает следующего посетителя и т.д. Если клиентов нет (комната ожидания пуста), парикмахер садится в свое рабочее кресло и засыпает. Будет его очередной появляющийся посетитель парикмахерской.

В данной задаче ресурсами являются места ожидания и рабочее кресло. Потоки-клиенты должны получить эти ресурсы строго последовательно: сначала посетитель должен найти место в комнате ожидания и только затем занять очередь к парикмахеру. При этом, предоставление рабочего кресла для обслуживания производит специальный процесс-парикмахер. В этом плане, парикмахера можно интерпретировать как сервер, предоставляющий требуемый сервис.

Для сравнения представим решение данной задачи при помощи монитора. Определим события, которые происходят в процессе вычислений (для каждого события сразу укажем условные переменные, которые будут использоваться для объявления этих событий):

- Client – событие, означающееся, что есть ожидающие посетители; событие объявляется каждый при появлении нового клиента; данное событие пробуждает спящего парикмахера, заснувшего при отсутствии клиентов;
- Barber – событие при освобождении парикмахера; по данному событию пробуждается один из ожидающих клиентов, который и переход в рабочую комнату для обслуживания;
- Service – событие при завершении обслуживание очередного клиента; клиент может покинуть парикмахерскую.

С учетом введенных условных переменных возможное решение задачи с использованием монитора может состоять в следующем.

```
Monitor BarberShop {  
    const int ClientMax = 10; // Максимальное количество посетителей  
    int ClientNum = 0; // Текущее количество клиентов  
    cond Client; // Событие: Имеются ожидающие посетители  
    cond Barber; // Событие: Парикмахер свободен  
    cond Service; // Событие: Обслуживание завершено  
  
    Client(){ // Поток клиента  
        if ( ClientNum < ClientMax ) { // Ждать только если есть места  
            ClientNum++;  
            signal(Client); // Есть посетители - необходимо разбудить парикмахера  
            wait(Barber); // Ждать парикмахера  
            wait(Service); // Ждать окончания стрижки  
        }  
    }  
  
    Barber(){ // Поток парикмахера  
        while (1) {  
            if ( ClientNum == 0 ) wait(Client); // Ждать посетителей (спать)  
            signal(Barber); // Парикмахер свободный - пригласить клиента  
            ClientNum--;  
            <Обслуживание>  
            signal(Service); // Обслуживание завершено - клиент может уйти  
        }  
    }  
}
```

Рассматривая данную задачу, попытайтесь разработать другие возможные способы решения (в частности, с использованием семафоров). Может быть также изменена или расширена постановка задачи (можно, например, увеличить количество парикмахеров).

Дополнительная информация по данной задаче может быть получена, например, в [33].

## **4.6. Методы повышения эффективности параллельных программ**

Соблюдение правил синхронизации, взаимоисключения и недопущения взаимоблокировки приводит к построению корректных параллельных программ. В данном подразделе будут кратко рассмотрены методы повышения эффективности параллельных вычислений.

### **4.6.1. Оптимизация количества потоков**

Количество используемых потоков существенным образом сказывается на эффективность параллельных вычислений. При малом количестве потоков (меньшим, чем число ядер/процессоров) не будет задействован полностью потенциал компьютерного оборудования. Однако избыток потоков также может негативно сказаться на эффективность и основные причины этого состоят в следующем:

- При большом количестве потоков могут увеличиться затраты на их обслуживание – потоки надо создавать и завершать; при нехватке вычислительных устройств (ядер/процессоров) для потоков нужно прерывать выполнение активных потоков, сохранять их состояния и передавать на выполнение новые потоки; следует отметить, что необходимое время на переключение потоков обычно является небольшим, а затраты на создание и завершение потоков можно снизить, если выполнять эти действия только при старте и завершении параллельной программ;
- Более серьезная причина потери эффективности может состоять в ухудшении продуктивного использования кэш-памяти; как известно, кэш-память является во много (10-100) раз быстрее обычной оперативной памяти и быстродействие программы можно значительно повысить, если обеспечить присутствие обрабатываемых данных в кэш-памяти; в ситуациях, когда из-за избытка потоков необходимо переключать вычислительные устройства (ядра/процессоры) на выполнение разных потоков, потоки должны восстанавливать состояние кэш-памяти при каждом своем новом возобновлении (что и приводит к появлению дополнительных задержек вычислений); аналогичная проблема возникает и при использовании виртуальной памяти (часть данных приостановленных потоков может быть вытеснена в медленную внешнюю память);
- Значительная проблема при большом количестве потоков состоит в существенном повышении затрат на организацию синхронизации и взаимоисключения потоков – как будет показано, например, в разделе 11 на примере задачи численного решения уравнений в частных производных, избыток потоков в силу чрезмерной синхронизации может привести не к ускорению, а к достаточно заметному замедлению вычислений; наихудшая ситуация может состоять в приостановке потоков, которые должны снять блокировку с общих разделяемых ресурсов, при переключении ядре/процессоров на выполнение новых потоков – в этом случае, потоки, которые активизированы для выполнения, не могут продолжаться из-за блокировки ресурсов, а потоки, которые могут снять эту блокировку, ждут своей очереди на выполнение.

Все высказанное говорит о том, что количество потоков должно выбираться тщательно. Обычно число потоков определяется как параметр запуска параллельной программы и, в лучшем случае, это параметр должен задаваться самой средой выполнения автоматически (например, средствами OpenMP). Выбор количества потоков должен осуществляться с учетом числа вычислительных элементов (ядер/процессоров), количеством устройств кэш-памяти (количество вычислительных элементов и устройств кэш-памяти могут не совпадать) и объемом выполняемых вычислений в потоках. В частности, целесообразно, чтобы количество вычислительно-интенсивных потоков совпадало с числом ядер/процессоров или с имеющимся количеством устройств кэш-памяти.

#### **4.6.2. Минимизация взаимодействия потоков**

Безусловно, максимальная эффективность параллельности достигается при полной независимости параллельно выполняемых потоков (при условии равного распределения вычислительной нагрузки). Любое взаимодействие и, как результат, синхронизация деятельности потоков приводит к появлению задержек и снижения быстродействия вычислений. Поскольку полностью избежать взаимодействия потоков невозможно (параллельные потоки занимается решением единой задачи), то общие рекомендации по снижению потерь от синхронизации состоят в следующем:

- Уменьшение, по мере возможности, количества необходимых взаимодействий потоков – так, например, если решается задача суммирование значений некоторого числового набора данных, то вместо использование единственной общей переменной для накопления суммы можно сначала вычислить частные суммы для каждого потока в отдельности (без каких-либо синхронизаций), а только потом собрать эти частные суммы вместе;
- Повышение эффективности алгоритмов, выполняемых в критических секциях потоков, и, тем самым, сокращение времени блокировки общих ресурсов – так, например, если в критической секции необходимо упорядочить данные, то, конечно же, необходимо использовать быстрые алгоритмы сортировки;
- Разделение общих ресурсов для разбиения единственной критической секции на множество раздельных и более редко используемых подсекций – так, при использовании таблицы данных можно организовать блокировку для каждой строки таблицы в отдельности;
- Обеспечение более быстрого выполнения потоков, которые находятся в своих критических секциях – для этого, в частности, можно запретить приостановку таких потоков или повысить их приоритет; применение таких методов обеспечивается обычно средствами выполнения параллельных программ.

#### **4.6.3. Оптимизация работы с памятью**

Обеспечение эффективного использования памяти является общей проблемой программирования. И ключевой момент в этой проблеме – оптимизация работы с кэш-памятью, поскольку время доступа в этой памяти существенно (в 10-100 раз) меньше по сравнению с оперативной памятью. Основной способ достижения эффективности – обеспечение локальности использования данных, когда информация после считывания в кэш-память многократно используется без обращения к медленной оперативной памяти. При этом следует учитывать, что перемещение данных в кэш осуществляется небольшими блоками – *строками* (*cache line*). Размер строк кэш-памяти обычно 64-128 байт. Как результат, после считывания некоторого значения в кэш, соседние элементы также оказываются в кэш и их обработка уже не потребует доступа к оперативной памяти. Учет этого момента также позволяет значительно повысить эффективность работы программы – так, например, при работе с матрицами данных более эффективно проводить обработку элементов по строкам, а не по столбцам (данное утверждение справедливо для алгоритмического языка С, в котором матрицы располагаются в памяти по строкам).

Полное рассмотрение вопросов оптимизации использования памяти выходит за пределы данного издания – дополнительная информация по данной проблеме может быть получена, например, в [16]. Пример решения вопросов оптимизации работы с памятью для блочного алгоритма умножения матриц рассмотрен в Главе 7 данного издания (следует отметить, что подобная оптимизация позволила уменьшить время работы алгоритма более чем в два раза).

При параллельном программировании для систем с общей памятью возникают дополнительные аспекты эффективного использования памяти, связанные с перемещением данных между имеющимися несколькими устройствами кэш-памяти.

### ***Обеспечение однозначности кэш-памяти***

Первый дополнительный аспект – это упоминавшаяся уже в Главе 1 проблема обеспечения *однозначности (когерентности)* памяти при наличии в системе нескольких устройств кэш-памяти. Напомним суть проблемы – при изменении каким-либо вычислительным ядром/процессором значения общей переменной, копии которой находятся в нескольких устройствах кэш-памяти, необходимо обновить значение этой переменной для всех ее копий (или запретить использование "устаревших" копий). Данный аспект связан, скорее всего, не с эффективностью, а с корректностью выполнения параллельных программ, и, обычно, обеспечивается на аппаратном уровне.

### ***Уменьшение миграции потоков между ядрами/процессорами***

Другой дополнительный аспект состоит в возможности возникновения дополнительных перемещений данных между разными устройствами кэш-памяти при смене вычислительного ядра/процессора для выполнения потоков. Для устранения этого эффекта в современных системах обычно имеются средства для обеспечения связности (*processor affinity*) потоков и используемых для их выполнения вычислительных устройств. Однако использование таких средств является достаточно не простым делом, поскольку одновременно необходимо обеспечить и масштабируемость вычислений (эффективность выполнения для разных конфигурациях имеющихся вычислительных ресурсов). Более простой способ снижения подобного эффекта может состоять в использовании количества потоков совпадающим с числом имеющихся вычислительных устройств (см. также п. 4.6.1).

### ***Устранение эффекта ложного разделения данных***

Еще один очень интересный аспект, связанный с наличием нескольких устройств кэш-памяти, состоит в возможности появления так называемого эффекта *ложного разделения данных (false sharing)*, когда части одной и той же строки кэш-памяти оказываются разделенными между разными кэшами. В этом случае любое изменение данных приводит к необходимости передачи строк кэш-памяти между устройствами кэш-памяти (однозначность памяти обеспечивается не на уровне отдельных переменных, а для полных строк кэш-памяти). Подобная синхронизация является излишней, если обрабатываемые данные на разных ядрах/процессорах не пересекаются. Возникновение такого эффекта может существенно снизить эффективность вычислений и для его недопущения достаточно обеспечить несмежное размещение в памяти данных, обрабатываемых разными вычислительными устройствами.

#### **4.6.4. Использование потоко-ориентированных библиотек**

В завершение рассмотрения вопросов повышения эффективности многопоточных параллельных программ общая рекомендация по полезности максимально-возможного использования программных библиотек, специально разработанных для вычислительных систем с общей памятью. Использование таких библиотек может быть и обязательным условием для корректности выполнения многопоточных программ – так, например, при построении программы компилятор должен использовать потоко-ориентированные версии служебных программ среды выполнения (необходимость поддержки многопоточности указывается компилятору при помощи соответствующих управляющих ключей). А, с другой стороны, применение уже имеющихся многопоточных библиотек крайне полезно – как правило, имеющиеся в этих библиотеках реализации параллельных

методов является высокоеффективными. И, кроме того, использование библиотек позволяет существенно снизить затраты на разработку необходимого параллельного программного обеспечения.

## 4.7. Краткий обзор главы

Данная глава посвящена рассмотрению основных аспектов параллельного программирования.

В 4.1 рассматривается ряд понятий и определений, являющихся основополагающими для параллельного программирования. Среди таких понятий концепция *процессов, потоков и ресурсов*. С использованием введенных понятий *параллельные программы* могут быть представлены как *системы параллельно выполняемых процессов и потоков*.

В 4.2 проводится последовательное рассмотрение возможных способов организации взаимоисключения параллельно выполняемых потоков. Первоначально в подразделе даются «очевидные», на первый взгляд, решения, но которые, на самом деле, являются неполными и позволяют продемонстрировать ряд проблем, которые могут возникать при разработке параллельных программ – *жесткая синхронизация, потеря взаимоисключения, возможность блокировки, бесконечное откладывание*. Далее приводится полное решение задачи взаимоисключения – *алгоритм Деккера*. И в завершение в подразделе рассматриваются классические механизмы организации взаимоисключения – *семафоры Дейстры и мониторы Хоара*.

В 4.3 проводится изучение проблемы синхронизации параллельно выполняемых потоков, для решения которой приводятся два основных наиболее широко используемых подхода – использование *условных переменных* и организация *барьерной синхронизации*.

В 4.4 рассматривается одна из основных проблем параллельного программирования – возникновения в ходе параллельных вычислений ситуаций *взаимоблокировки* потоков, когда потоки не могут продолжить свое выполнение из-за конкуренции за общие разделяемые ресурсы (такие ситуации в литературе называются также как *туники, дедлоки или смертельные объятия*). При изучении проблемы определяются условия возникновения тупиковых ситуаций. Далее рассматривается модель программы в виде *графа «поток-ресурс»*, которая позволяет анализировать процесс выполнения параллельных программ и определять наличие условий возникновения тупиков.

В 4.5 рассматривается ряд иллюстративных примеров, которые принято считать в качестве классических задач параллельного программирования, поскольку позволяют продемонстрировать многие проблемы, возникающие при разработке параллельных алгоритмов и программ, и предоставляют возможность наглядно показать основные способы решения этих проблем. В числе этих задач:

- Задача "Производители-Потребители" (*Producer-Consumer problem*);
- Задача "Читатели-Писатели" (*Readers-Writers problem*);
- Задача "Обедающие философы" (*Dining Philosopher problem*);
- Задача "Спящий брадобрей" (*Sleeping Barber problem*).

В 4.6 излагается ряд практических методов повышения эффективности параллельных программ. К числу рассматриваемых методов относится оптимизация количества потоков, минимизация взаимодействия потоков, оптимизация работы с памятью и широкое использование ранее разработанных библиотек параллельных методов.

## 4.8. Обзор литературы

Дополнительная информация по вопросам, изложенным в данном разделе, может быть получена, например, в [39]. Проблемы параллельного программирования

применительно к тематике операционных систем широко рассмотрены в [5,15,27,31-33]. Вопросы моделирования процессов выполнения параллельных программ излагаются в [36].

Для рассмотрения вопросов организации параллельных программ применительно к операционной системе Windows могут быть рекомендованы работы [22,25], для изучения этих же вопросов для ОС Unix может быть рекомендована работа [26].

## 4.9. Контрольные вопросы

1. В чем состоят понятия процесса и потока? Укажите схожесть и различия этих понятий.
2. В чем состоит понятие ресурса? Приведите примеры различных типов ресурсов.
3. Дайте общую характеристику представления параллельных программ как системы параллельно выполняемых потоков?
4. Какие основные предположения могут быть сделаны о характере временных соотношений между выполняемыми командными последовательностями разных потоков?
5. Чем определяется повышенная сложность параллельного программирования?
6. В чем состоит проблема взаимоисключения потоков? Какие основные требования к методам решения этой проблемы?
7. В чем состоит недостаток метода жесткой синхронизации при организации взаимоисключения потоков?
8. Приведите примеры некорректного решения проблемы взаимоисключения потоков, при котором происходит потеря взаимоисключения?
9. Приведите примеры некорректного решения проблемы взаимоисключения потоков, при котором возможно возникновение взаимоблокировки потоков?
10. Приведите примеры некорректного решения проблемы взаимоисключения потоков, при котором возможна ситуация бесконечного откладывания доступа к критическим секциям?
11. В чем состоит алгоритм Деккера для решения проблемы взаимоисключения потоков?
12. В чем состоит концепция семафоров Дейкстры? Приведите пример решения проблемы взаимоисключения потоков с использованием семафоров.
13. В чем состоит концепция мониторов Хоара? Приведите пример решения проблемы взаимоисключения потоков с использованием мониторов.
14. В чем состоит проблема синхронизации потоков?
15. Как решается проблема синхронизации потоков при помощи условных переменных?
16. Как решается проблема синхронизации потоков при помощи метода барьерной синхронизации?
17. В чем состоит проблема взаимоблокировки потоков? Укажите необходимые условия возникновения тупиков.
18. В чем состоит модель параллельных программ в виде графа «поток-ресурс»?
19. Какие свойства параллельных программ могут быть получены в результате анализа графа «поток-ресурс»?
20. Дайте общую характеристику и приведите возможное решение задачи "Производители-Потребители" (Producer-Consumer problem).
21. Дайте общую характеристику и приведите возможное решение задачи "Читатели-Писатели" (Readers-Writers problem).

22. Дайте общую характеристику и приведите возможное решение задачи "Обедающие философы" (Dining Philosopher problem).
23. Дайте общую характеристику и приведите возможное решение задачи "Спящий брадобрей" (Sleeping Barber problem).
24. Каким образом оптимизация количества потоков может повысить эффективность выполнения параллельных программ?
25. Каким образом минимизация взаимодействия потоков может повысить эффективность выполнения параллельных программ?
26. Каким образом оптимизация работы с памятью может повысить эффективность выполнения параллельных программ?
27. Каким образом использование библиотек параллельных методов может повысить эффективность выполнения параллельных программ?

## 4.10. Задачи и упражнения

1. Изучите методы синхронизации и взаимоисключения потоков для операционной системы Windows и разработайте ряд демонстрационных параллельных программ.
2. Изучите методы синхронизации и взаимоисключения потоков для операционной системы Unix/Linux и разработайте ряд демонстрационных параллельных программ.
3. Изучите методы синхронизации и взаимоисключения потоков для стандарта POSIX.
4. Изучите методы синхронизации и взаимоисключения потоков для технологии OpenMP.
5. Разработайте несколько параллельных программ для решения задачи "Производители-Потребители" с использованием разных механизмов синхронизации и взаимоисключения потоков.
6. Разработайте несколько параллельных программ для решения задачи "Читатели-Писатели" с использованием разных механизмов синхронизации и взаимоисключения потоков.
7. Разработайте несколько параллельных программ для решения задачи "Обедающие философы" с использованием разных механизмов синхронизации и взаимоисключения потоков.
8. Разработайте несколько параллельных программ для решения задачи "Спящий брадобрей" с использованием разных механизмов синхронизации и взаимоисключения потоков.
9. Разработайте несколько параллельных программ для демонстрации способов оптимизации количества потоков для повышения эффективности выполнения параллельных программ.
10. Разработайте несколько параллельных программ для демонстрации способов минимизации взаимодействия потоков для повышения эффективности выполнения параллельных программ.
11. Разработайте несколько параллельных программ для демонстрации способов оптимизации работы с памятью для повышения эффективности выполнения параллельных программ.
12. Разработайте несколько параллельных программ для демонстрации результативности использования библиотек параллельных методов для повышения эффективности выполнения параллельных программ.

Глава 5 . Параллельное программирование с использованием OpenMP .....	2
5.1. Основы технологии OpenMP .....	4
5.1.1. Понятие параллельной программы.....	4
5.1.2. Организация взаимодействия параллельных потоков .....	5
5.1.3. Структура OpenMP .....	6
5.1.4. Формат директив OpenMP .....	6
5.2. Выделение параллельно-выполняемых фрагментов программного кода.....	7
5.2.1. Директива parallel для определения параллельных фрагментов.....	7
5.2.2. Пример первой параллельной программы.....	7
5.2.3. Основные понятия параллельной программы: фрагмент, область, секция.....	8
5.2.4. Параметры директивы parallel.....	9
5.2.5. Определение времени выполнения параллельной программы.....	9
5.3. Распределение вычислительной нагрузки между потоками (распараллеливание по данным для циклов) .....	10
5.3.1. Управление распределением итераций цикла между потоками.....	12
5.3.2. Управление порядком выполнения вычислений.....	13
5.3.3. Синхронизация вычислений по окончании выполнения цикла.....	13
5.3.4. Введение условий при определении параллельных фрагментов (параметр if директивы parallel).....	14
5.4. Управление данными для параллельно-выполняемых потоков .....	14
5.4.1. Определение общих и локальных переменных.....	15
5.4.2. Совместная обработка локальных переменных (операция редукции) .....	16
5.5. Организация взаимоисключения при использовании общих переменных .....	17
5.5.1. Обеспечение атомарности (неделимости) операций .....	17
5.5.2. Использование критических секций .....	18
5.5.3. Применение переменных семафорного типа (замков).....	19
5.6. Распределение вычислительной нагрузки между потоками (распараллеливание по задачам при помощи директивы sections) .....	20
5.7. Расширенные возможности OpenMP .....	22
5.7.1. Определение однопотоковых участков для параллельных фрагментов (директивы single и master).....	22
5.7.2. Выполнение барьерной синхронизации (директива barrier) .....	23
5.7.3. Синхронизация состояния памяти (директива flush).....	23
5.7.4. Определение постоянных локальных переменных потоков (директива threadprivate и параметр copyin директивы parallel) .....	24
5.7.5. Управление количеством потоков .....	25
5.7.6. Задание динамического режима при создании потоков .....	26
5.7.7. Управление вложенностью параллельных фрагментов .....	26
5.8. Дополнительные сведения .....	26
5.8.1. Разработка параллельных программ с использованием OpenMP на алгоритмическом языке Fortran.....	27
5.8.2. Сохранение возможности последовательного выполнения программы.....	28
5.8.3. Краткий перечень компиляторов с поддержкой OpenMP .....	29
5.9. Краткий обзор раздела .....	29
5.10. Обзор литературы .....	30
5.11. Контрольные вопросы .....	31
Задачи и упражнения.....	32
Приложение: Справочные сведения об OpenMP .....	33
П1. Сводный перечень директив OpenMP .....	33
П2. Сводный перечень параметров директив OpenMP .....	34

П3. Сводный перечень функций OpenMP.....	36
П4. Сводный перечень переменных окружения OpenMP.....	37

## Глава 5.

### Параллельное программирование с использованием OpenMP

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередьность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (*symmetric multiprocessors, SMP*).

Перечисленному выше набору предположений удовлетворяют также активно развивающиеся в последнее время *многоядерные процессоры*, в которых каждое ядро представляет практически независимо функционирующее вычислительное устройство. Для общности излагаемого учебного материала для упоминания одновременно и мультипроцессоров и многоядерных процессоров для обозначения одного вычислительного устройства (одноядерного процессора или одного процессорного ядра) будет использоваться понятие *вычислительного элемента (ВЭ)*.

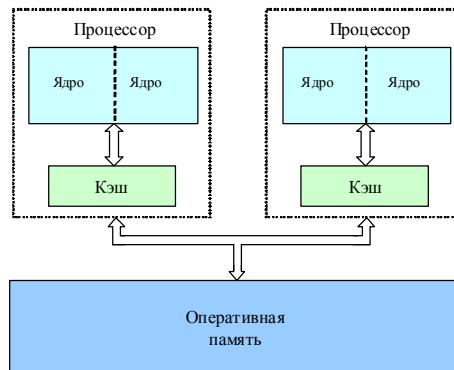


Рис. 5.1. Архитектура многопроцессорных систем с общей (разделяемой) с однородным доступом памятью (для примера каждый процессор имеет два вычислительных ядра)

Следует отметить, что общий доступ к данным может быть обеспечен и при физически分散된 памяти (при этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти). Такой подход именуется как *неоднородный доступ к памяти (non-uniform memory access or NUMA)*.

В самом общем виде системы с общей памятью могут быть представлены в виде модели параллельного компьютера с произвольным доступом к памяти (*parallel random-access machine – PRAM*) - см., например, [39].

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное

программирование, так и уже имеющиеся языки программирования, расширенные некоторым набором операторов для параллельных вычислений.

Широко используемый подход состоит и в применении тех или иных библиотек, обеспечивающих определенный программный интерфейс (*application programming interface, API*) для разработки параллельных программ. В рамках такого подхода наиболее известны Windows Thread API (см., например, [7]) и PThread API (см., например, [46]). Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоемким для использования и имеет низкоуровневый характер.

Все перечисленные выше подходы приводят к необходимости существенной переработки существующего программного обеспечения, и это в значительной степени затрудняет широкое распространение параллельных вычислений. Как результат, в последнее время активно развивается еще один подход к разработке параллельных программ, когда указания программиста по организации параллельных вычислений добавляются в программу при помощи тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный текст программы остается неизменным, и по нему, в случае отсутствия препроцессора, компилятор построит исходный последовательный программный код. Препроцессор же, будучи примененным, заменяет директивы параллелизма на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки).

Рассмотренный выше подход является основой технологии *OpenMP* (см., например, [48]), наиболее широко применяемой в настоящее время для организации параллельных вычислений на многопроцессорных системах с общей памятью. В рамках данной технологии директивы параллелизма используются для выделения в программе *параллельных фрагментов*, в которых последовательный исполняемый код может быть разделен на несколько раздельных командных потоков (*threads*). Далее эти потоки могут исполняться на разных процессорах (процессорных ядрах) вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (*однопоточных*) и параллельных (*многопоточных*) участков программного кода (см. рис. 5.2). Подобный принцип организации параллелизма получил наименование "вилочного" (*fork-join*) или *пульсирующего параллелизма*. Более полная информация по технологии OpenMP может быть получена в литературе (см., например, [1, 48, 85]) или в информационных ресурсах сети Интернет.

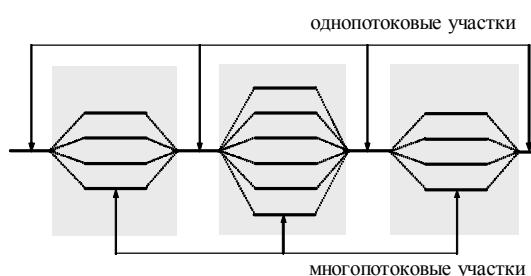


Рис. 5.2. Общая схема выполнения параллельной программы при использовании технологии OpenMP

При разработке технологии OpenMP был учтен накопленный опыт по разработке параллельных программ для систем с общей памятью. Опираясь на стандарт X3Y5 (см. [48]) и учитывая возможности PThreads API (см. [46]), в технологии OpenMP в значительной степени упрощена форма записи директив и добавлены новые функциональные возможности. Для привлечения к разработке OpenMP самых опытных

специалистов и для стандартизации подхода на самых ранних этапах выполнения работ был сформирован Международный комитет по OpenMP (*the OpenMP Architectural Review Board, ARB*). Первый стандарт, определяющий технологию OpenMP применительно к языку Fortran, был принят в 1997 г., для алгоритмического языка С – в 1998 г. Последняя версия стандарта OpenMP для языков С и Fortran была опубликована в 2005 г. (см. [www.openmp.org](http://www.openmp.org)).

Далее в Главе будет приведено последовательное описание возможностей технологии OpenMP. Здесь же, еще не приступая к изучению, приведем ряд важных положительных моментов этой технологии:

- Технология OpenMP позволяет в максимальной степени эффективно реализовать возможности многопроцессорных вычислительных систем с общей памятью, обеспечивая использование общих данных для параллельно выполняемых потоков без каких-либо трудоемких межпроцессорных передач сообщений.
- Сложность разработки параллельной программы с использованием технологии OpenMP в значительной степени согласуется со сложностью решаемой задачи – распараллеливание сравнительно простых последовательных программ, как правило, не требует больших усилий (порою достаточно включить в последовательную программу всего лишь несколько директив OpenMP)<sup>1)</sup>; это позволяет, в частности, разрабатывать параллельные программы и прикладным разработчикам, не имеющим большого опыта в параллельном программировании.
- Технология OpenMP обеспечивает возможность поэтапной (*инкрементной*) разработки параллельных программы – директивы OpenMP могут добавляться в последовательную программу постепенно (поэтапно), позволяя уже на ранних этапах разработки получать параллельные программы, готовые к применению; при этом важно отметить, что программный код получаемых последовательного и параллельного вариантов программы является единым и это в значительной степени упрощает проблему сопровождения, развития и совершенствования программ.
- OpenMP позволяет в значительной степени снизить остроту проблемы переносимости параллельных программ между разными компьютерными системами – параллельная программа, разработанная на алгоритмическом языке С или Fortran с использованием технологии OpenMP, как правило, будет работать для разных вычислительных систем с общей памятью.

## 5.1. Основы технологии OpenMP

Перед началом практического изучения технологии OpenMP рассмотрим ряд основных понятий и определений, являющихся основополагающими для данной технологии.

### 5.1.1. Понятие параллельной программы

Под *параллельной программой* в рамках OpenMP понимается программа, для которой в специально указываемых при помощи директив местах – *параллельных фрагментах* – исполняемый программный код может быть разделен на несколько раздельных командных *потоков* (*threads*). В общем виде программа представляется в виде набора последовательных (*однопоточных*) и параллельных (*многопоточных*) участков программного кода (см. рис. 5.2).

---

<sup>1)</sup> Сразу хотим предупредить, чтобы простота применения OpenMP для первых простых программ не должна приводить в заблуждение – при разработке сложных алгоритмов и программ требуется соответствующий уровень усилий и для организации параллельности – см. раздел 11 настоящего учебного материала.

Важно отметить, что разделение вычислений между потоками осуществляется под управлением соответствующих директив OpenMP. Равномерное распределение вычислительной нагрузки – *балансировка (load balancing)* – имеет принципиальное значение для получения максимально возможного ускорения выполнения параллельной программы.

Потоки могут выполняться на разных процессорах (процессорных ядрах) либо могут группироваться для исполнения на одном вычислительном элементе (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Количество потоков определяется в начале выполнения параллельных фрагментов программы и обычно совпадает с количеством имеющихся вычислительных элементов в системе; изменение количества создаваемых потоков может быть выполнено при помощи целого ряда средств OpenMP. Все потоки в параллельных фрагментах программы последовательно перенумерованы от 0 до  $pr-1$ , где  $pr$  есть общее количество потоков. Номер потока также может быть получен при помощи функции OpenMP.

Использование в технологии OpenMP потоков для организации параллелизма позволяет учесть преимущества многопроцессорных вычислительных систем с общей памятью. Прежде всего, потоки одной и той же параллельной программы выполняются в общем адресном пространстве, что обеспечивает возможность использования общих данных для параллельно выполняемых потоков без каких-либо трудоемких межпроцессорных передач сообщений (в отличие от процессов в технологии MPI для систем с распределенной памятью). И, кроме того, управление потоками (создание, приостановка, активизация, завершение) требует меньше накладных расходов для ОС по сравнению с процессами.

### 5.1.2. Организация взаимодействия параллельных потоков

Как уже отмечалось ранее, потоки исполняются в общем адресном пространстве параллельной программы. Как результат, взаимодействие параллельных потоков можно организовать через использование общих данных, являющихся доступными для всех потоков. Наиболее простая ситуация состоит в использовании общих данных только для чтения. В случае же, когда общие данные могут изменяться несколькими потоками, необходимы специальные усилия для организации правильного взаимодействия. На самом деле, пусть два потока исполняют один и тот же программный код

```
n=n+1;
```

для общей переменной  $n$ . Тогда в зависимости от условий выполнения данная операция может быть выполнена поочередно (что приведет к получению правильного результата) или же оба потока могут одновременно прочитать значение переменной  $n$ , одновременно увеличить и записать в эту переменную новое значение (как результат, будет получено неправильное значение). Подобная ситуация, когда результат вычислений зависит от темпа выполнения потоков, получил наименование *гонки потоков (race conditions)*. Для исключения гонки необходимо обеспечить, чтобы изменение значений общих переменных осуществлялось в каждый момент времени только одним единственным потоком – иными словами, необходимо обеспечить *взаимное исключение (mutual exclusion)* потоков при работе с общими данными. В OpenMP взаимоисключение может быть организовано при помощи *неделимых (atomic)* операций, механизма *критических секций (critical sections)* или специального типа семафоров – *замков (locks)*.

Следует отметить, что организация взаимного исключения приводит к уменьшению возможности параллельного выполнения потоков – при одновременном доступе к общим переменным только один из них может продолжить работу, все остальные потоки будут блокированы и будут ожидать освобождения общих данных. Можно сказать, что именно при реализации взаимодействия потоков проявляется искусство параллельного программирования для вычислительных систем с общей памятью – организация взаимного исключения при работе с общими данными является обязательной, но возникающие при этом задержки (блокировки) потоков должны быть минимальными по времени.

Помимо взаимоисключения, при параллельном выполнении программы во многих случаях является необходимым та или иная *синхронизация* (*synchronization*) вычислений, выполняемых в разных потоках: например, обработка данных, выполняемая в одном потоке, может быть начата только после того, как эти данные будут сформированы в другом потоке (классическая задача параллельного программирования "производитель-потребитель" – "*producer-consumer*" problem). В OpenMP синхронизация может быть обеспечена при помощи замков или директивы **barrier**.

### 5.1.3. Структура OpenMP

Конструктивно в составе технологии OpenMP можно выделить:

- Директивы,
- Библиотеку функций,
- Набор переменных окружения.

Именно в таком порядке и будут рассмотрены возможности технологии OpenMP.

### 5.1.4. Формат директив OpenMP

Стандарт предусматривает использование OpenMP для алгоритмических языков C90, C99, C++, Fortran 77, Fortran 90 и Fortran 95. Далее описание формата директив OpenMP и все приводимые примеры программ будут представлены на алгоритмическом языке C; особенности использования технологии OpenMP для алгоритмического языка Fortran будут даны в п. 5.8.1.

В самом общем виде формат директив OpenMP может быть представлен в следующем виде:

```
#pragma omp <имя_директивы> [<параметр>[ [ , ] <параметр>]...]
```

Начальная часть директивы (#pragma omp) является фиксированной, вид директивы определяется ее именем (имя\_директивы), каждая директива может сопровождаться произвольным количеством параметров (на английском языке для параметров директивы OpenMP используется термин *clause*).

Для иллюстрации приведем пример директивы:

```
#pragma omp parallel default(shared) \
                      private(beta,pi)
```

Пример показывает также, что для задания директивы может быть использовано несколько строк программы – признаком наличия продолжения является знак обратного слеша "\".

Действие директивы распространяется, как правило, на следующий в программе оператор, который может быть, в том числе, и структурированным блоком.

## 5.2. Выделение параллельно-выполняемых фрагментов программного кода

Итак, параллельная программа, разработанная с использованием OpenMP, представляется в виде набора последовательных (однопотоковых) и параллельных (многопотоковых) фрагментов программного кода (см. рис. 5.2).

### 5.2.1. Директива **parallel** для определения параллельных фрагментов

Для выделения параллельных фрагментов программы следует использовать директиву **parallel**:

```
#pragma omp parallel [<параметр> ...]  
    <блок_программы>
```

Для блока (как и для блоков всех других директив OpenMP) должно выполняться правило "один вход – один выход", т.е. передача управления извне в блок и из блока за пределы блока не допускается.

Директива **parallel** является одной из основных директив OpenMP. Правила, определяющие действия директивы, состоят в следующем:

- Когда программа достигает директиву **parallel**, создается набор (*team*) из *N* потоков; исходный поток программы является основным потоком этого набора (*master thread*) и имеет номер 0.
- Программный код блока, следующий за директивой, дублируется или может быть разделен при помощи директив между потоками для параллельного выполнения.
- В конце программного блока директивы обеспечивается синхронизация потоков – выполняется ожидание окончания вычислений всех потоков; далее все потоки завершаются – дальнейшие вычисления продолжает выполнять только основной поток (в зависимости от среды реализации OpenMP потоки могут не завершаться, а приостанавливаться до начала следующего параллельного фрагмента – такой подход позволяет снизить затраты на создание и удаление потоков).

### 5.2.2. Пример первой параллельной программы

Подчеркнем чрезвычайно важный момент – оказывается, даже такого краткого рассмотрения возможностей технологии OpenMP достаточно для разработки пусть и простых, но параллельных программ. Приведем практически стандартную первую программу, разрабатываемую при освоению новых языков программирования – программу, осуществляющую вывод приветственного сообщения "Hello World !" Итак:

```
#include <omp.h>  
main () {  
    /* Выделение параллельного фрагмента */  
    #pragma omp parallel  
    {  
        printf("Hello World !\n");  
    }/* Завершение параллельного фрагмента */  
}
```

Пример 5.1. Первая параллельная программа на OpenMP

В приведенной программе файл *omp.h* содержит определения именованных констант, прототипов функций и типов данных OpenMP – подключение этого файла является обязательным, если в программе используются функции OpenMP. При выполнении программы по директиве **parallel** будут созданы потоки (по умолчанию их количество совпадает с числом имеющихся вычислительных элементов системы –

процессоров или ядер), каждый поток выполнит программный блок, следуемый за директивой, и, как результат, программа выведет сообщение "Hello World !" столько раз, сколько будет иметься потоков.

### 5.2.3. Основные понятия параллельной программы: фрагмент, область, секция

После рассмотрения примера важно отметить также, что параллельное выполнение программы будет осуществляться не только для программного блока, непосредственно следующего за директивой **parallel**, но и для всех функций, вызываемых из этого блока (см. рис. 5.3). Для обозначения этих динамически-возникающих параллельно выполняемых участков программного кода в OpenMP используется понятие *параллельных областей* (*parallel region*) – ранее, в предшествующих версиях стандарта использовался термин *динамический контекст* (*dynamic extent*).

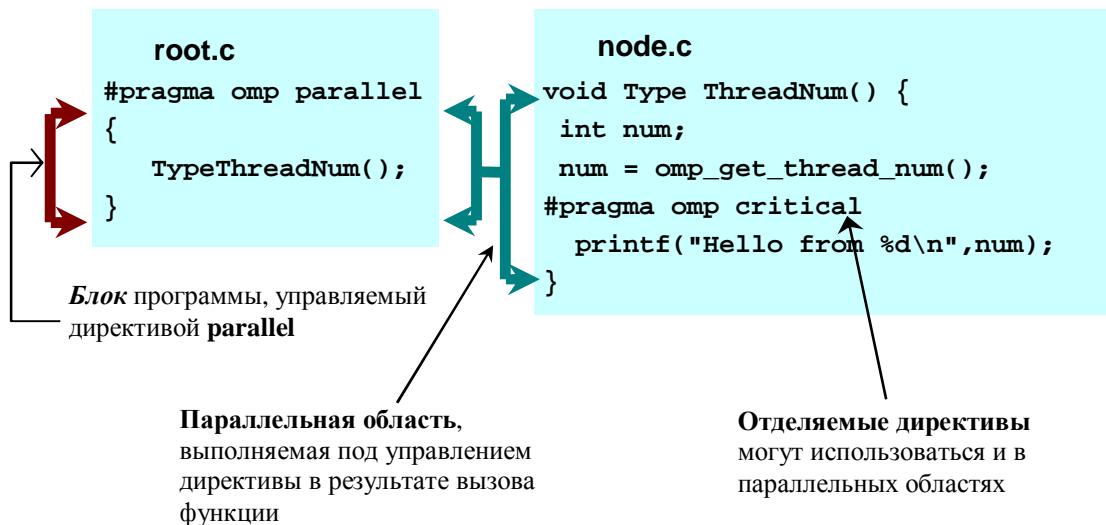


Рис. 5.3. Область видимости директив OpenMP

Ряд директив OpenMP допускает использование как непосредственно в блоках директивы, так и в параллельных областях. Такие директивы носят наименование *отделяемых директив* (*orphaned directives*).

Для более точного понимания излагаемого учебного материала сведем воедино введенные термины и понятия (в скобках даются названия, используемые в стандарте 2.5):

- *Параллельный фрагмент* (*parallel construct*) – блок программы, управляемый директивой **parallel**; именно параллельные фрагменты, совместно с параллельными областями, представляют параллельно-выполняемую часть программы; в предшествующих стандартах для обозначения данного понятия использовался термин *лексический контекст* (*lexical extent*) директивы **parallel**.
- *Параллельная область* (*parallel region*) – параллельно выполняемые участки программного кода, динамически-возникающие в результате вызова функций из параллельных фрагментов – см. рис. 5.3.
- *Параллельная секция* (*parallel section*) – часть параллельного фрагмента, выделяемая для параллельного выполнения при помощи директивы **section** – см. подраздел 5.6.

#### 5.2.4. Параметры директивы parallel

Приведем перечень параметров директивы **parallel**:

- if (scalar\_expression)
- private (list)
- shared (list)
- default (shared | none)
- firstprivate (list)
- reduction (operator: list)
- copyin (list)
- num\_threads (scalar\_expression)

Список параметров приведен только для справки, пояснения по использованию этих параметров будут даны позднее по мере изложения учебного материала.

#### 5.2.5. Определение времени выполнения параллельной программы

Практически сразу после разработки первой параллельной программы появляется необходимость определения времени выполнения вычислений, поскольку в большинстве случаев основной целью использования параллельных вычислительных систем является сокращение времени выполняемых расчетов. Используемые обычно средства для измерения времени работы программ зависят, как правило, от аппаратной платформы, операционной системы, алгоритмического языка и т.п. Стандарт OpenMP включает определение специальных функций для измерения времени, использование которых позволяет устранить зависимость от среды выполнения параллельных программ.

Получение текущего момента времени выполнения программы обеспечивается при помощи функции:

```
double omp_get_wtime(void),
```

результат вызова которой есть количество секунд, прошедших от некоторого определенного момента времени в прошлом. Этот момент времени в прошлом, от которого происходит отсчет секунд, может зависеть от среды реализации OpenMP и, тем самым, для ухода от такой зависимости функцию *omp\_get\_wtime* следует использовать только для определения длительности выполнения тех или иных фрагментов кода параллельных программ. Возможная схема применения функции *omp\_get\_wtime* может состоять в следующем:

```
double t1, t2, dt;  
t1 = omp_get_wtime();  
...  
t2 = omp_get_wtime();  
dt = t2 - t1;
```

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция:

```
double omp_get_wtick(void),
```

позволяющая определить время в секундах между двумя последовательными показателями времени аппаратного таймера используемой компьютерной системы.

### 5.3. Распределение вычислительной нагрузки между потоками (распараллеливание по данным для циклов)

Как уже отмечалось ранее, программный код блока директивы **parallel** по умолчанию исполняется всеми потоками. Данный способ может быть полезен, когда нужно выполнить одни и те же действия многократно (как в примере 5.1) или когда один и тот же программный код может быть применен для выполнения обработки разных данных. Последний вариант достаточно широко используется при разработке параллельных алгоритмов и программ и обычно именуется *распараллеливанием по данным*. В рамках данного подхода в OpenMP наряду с обычным повторением в потоках одного и того же программного кода – как в директиве **parallel** – можно осуществить разделение итеративно-выполняемых действий в циклах для непосредственного указания, над какими данными должны выполняться соответствующие вычисления. Такая возможность является тем более важной, поскольку во многих случаях именно в циклах выполняется основная часть вычислительно-трудоемких вычислений.

Для распараллеливания циклов в OpenMP применяется директива **for**:

```
#pragma omp for [<параметр> ...]  
    <цикл_for>
```

После этой директивы итерации цикла распределяются между потоками и, как результат, могут быть выполнены параллельно (см. рис. 5.4) – понятно, что такое распараллеливание возможно только, если между итерациями цикла нет информационной зависимости.

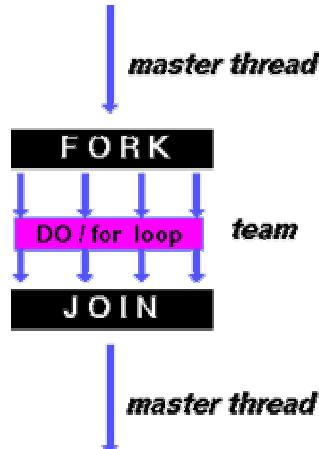


Рис. 5.4. Общая схема распараллеливания циклов

Важно отметить, что для распараллеливания цикл *for* должен иметь некоторый "канонический" тип цикла со счетчиком<sup>2)</sup>:

```
for (index = first; index < end; increment_expr)
```

Здесь *index* должен быть целой переменной; на месте знака "<" в выражении для проверки окончания цикла может находиться любая операция сравнения "<=", ">" или ">=". Операция изменения переменной цикла должна иметь одну из следующих форм:

- *index++*, *++index*,
- *index--*, *--index*,
- *index+=incr*, *index-=incr*,

<sup>2)</sup> Смысл требования "каноничности" состоит в том, чтобы на момент начала выполнения цикла существовала возможность определения числа итераций цикла

- `index=index+incr, index=incr+index,`
- `index=index-incr`

И, конечно же, переменные, используемые в заголовке оператора цикла, не должны изменяться в теле цикла.

В качестве примера использования директивы рассмотрим учебную задачу вычисления суммы элементов для каждой строки прямоугольной матрицы:

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () {
    int i, j, sum;
    float a[NMAX][NMAX];
    <инициализация данных>
#pragma omp parallel shared(a) private(i,j,sum)
{
    #pragma omp for
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=0; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n",i,sum);
    } /* Завершение параллельного фрагмента */
}
```

Пример 5.2. Пример распараллеливания цикла

В приведенной программе для директивы **parallel** появились два параметра – их назначение будет описано в следующем подразделе, здесь же отметим, что параметры директивы *shared* и *private* определяют доступность данных в потоках программы – переменные, описанные как *shared*, являются общими для потоков; для переменных с описанием *private* создаются отдельные копии для каждого потока, эти локальные копии могут использоваться в потоках независимо друг от друга.

Следует отметить, что если в блоке директивы **parallel** нет ничего, кроме директивы **for**, то обе директивы можно объединить в одну, т.е. пример 5.2 может быть переписан в виде:

```
#include <omp.h>
#define NMAX 1000
main () {
    int i, j, sum;
    float a[NMAX][NMAX];
    <инициализация данных>
#pragma omp parallel for shared(a) private(i,j,sum)
{
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=0; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n",i,sum);
    } /* Завершение параллельного фрагмента */
}
```

Пример 5.3. Пример использования объединенной директивы **parallel for**

Параметрами директивы **for** являются:

- `schedule (type [,chunk])`
- `ordered`
- `nowait`

- private (list)
- shared (list)
- firstprivate (list)
- lastprivate (list)
- reduction (operator: list)

Последние пять параметров директивы будут рассмотрены в следующем подразделе, здесь же приведем описание оставшихся параметров.

### 5.3.1. Управление распределением итераций цикла между потоками

При разном объеме вычислений в разных итерациях цикла желательно иметь возможность управлять распределением итераций цикла между потоками – в OpenMP это обеспечивается при помощи параметра *schedule* директивы **for**. Поле *type* параметра *schedule* может принимать следующие значения:

- **static** – статический способ распределения итераций до начала выполнения цикла. Если поле *chunk* не указано, то итерации делятся поровну между потоками. При заданном значении *chunk* итерации цикла делятся на блоки размера *chunk* и эти блоки распределяются между потоками до начала выполнения цикла.

- **dynamic** – динамический способ распределения итераций. До начала выполнения цикла потокам выделяются блоки итераций размера *chunk* (если поле *chunk* не указано, то полагается значение *chunk*=1). Дальнейшее выделение итераций (также блоками размера *chunk*) осуществляется в момент завершения потоками своих ранее назначенных итераций.

- **guided** – управляемый способ распределения итераций. Данный способ близок к предшествующему варианту, отличие состоит только в том, что начальный размер блоков итераций определяется в соответствии с некоторым параметром среды реализации OpenMP, а затем уменьшается экспоненциально (следующее значение *chunk* есть некоторая доля предшествующего значения) при каждом новом выделении блока итераций. При этом получаемый размер блока итераций не должен быть меньше значения *chunk* (если поле *chunk* не указано, то полагается значение *chunk*=1).

- **runtime** – способ распределения итераций<sup>3)</sup>, при котором выбор конкретной схемы (из ранее перечисленных) осуществляется в момент начала выполнения программы в соответствии со значением переменной окружения OMP\_SCHEDULE. Так, например, для задания динамического способа при размере блока итераций 3, следует определить:

```
setenv OMP_SCHEDULE "dynamic,3"
```

Полезность такого варианта очевидна – способ распределения итераций между потоками можно менять, не корректируя при этом код программы (т.е. без повторной компиляции и сборки программы).

Для демонстрации примера использования параметра *schedule* предположим, что матрица в примере 5.3 имеет верхний треугольный вид – в этом случае объем вычислений для каждой строки является различным и последовательное распределение итераций поровну приведет к неравномерному распределению вычислительной нагрузки между потоками. Для балансировки расчетов можно применить статическую или динамическую схемы распределения итераций:

```
#include <omp.h>
#define CHUNK 100
#define NMAX 1000
main () {
```

---

<sup>3)</sup> Поле *chunk* для способа **runtime** не применимо.

```

int i, j, sum;
float a[NMAX][NMAX];
<инициализация данных>
#pragma omp parallel for shared(a) private(i,j,sum) \
    schedule (dynamic, CHUNK)
{
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=i; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n",i,sum);
    } /* Завершение параллельного фрагмента */
}

```

Пример 5.4. Пример динамического распределения итераций между потоками (использование параметра *schedule* директивы **for**)

### 5.3.2. Управление порядком выполнения вычислений

В результате распараллеливания цикла порядок выполнения итераций не фиксирован: в зависимости от состояния среды выполнения очередность выполнения итераций может меняться. Если же для ряда действий в цикле необходимо сохранить первичный порядок вычислений, который соответствует последовательному выполнению итераций в последовательной программе, то желаемого результата можно добиться при помощи директивы **ordered** (при этом для директивы **for** должен быть указан параметр *ordered*). Поясним сказанное на примере нашей учебной задачи. Для приведенного выше варианта программы печать сумм элементов строк матрицы будет происходить в некотором произвольном порядке; при необходимости печати по порядку расположения строк следует провести следующее изменение программного кода:

```

#pragma omp parallel for shared(a) private(i,j,sum) \
    schedule (dynamic, CHUNK) ordered
{
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=i; j < NMAX; j++)
            sum += a[i][j];
    #pragma omp ordered
        printf ("Сумма элементов строки %d равна %f\n",i,sum);
    } /* Завершение параллельного фрагмента */
}

```

Пример 5.5. Пример использование директивы **ordered**

Поясним дополнительно, что параметр *ordered* управляет порядком выполнения только тех действий, которые выделены директивой **ordered** – выполнение оставшихся действий в итерациях цикла по-прежнему может происходить параллельно. Важно помнить также, что директива **ordered** может быть применена в теле цикла только один раз.

Следует отметить, что указание необходимости сохранения порядка вычислений может привести к задержкам при параллельном выполнении итераций, что ограничит возможность получения максимального ускорения вычислений.

### 5.3.3. Синхронизация вычислений по окончании выполнения цикла

По умолчанию, все потоки, прежде чем перейти к выполнению дальнейших вычислений, ожидают окончания выполнения итераций цикла даже если некоторые из них уже завершили свои вычисления – конец цикла представляет собой некоторый

барьер, который потоки могут преодолеть только все вместе. Можно отменить указанную синхронизацию, указав параметр *nowait* в директиве **for** – тогда потоки могут продолжить вычисления за переделами цикла, если для них нет итераций цикла для выполнения.

#### 5.3.4. Введение условий при определении параллельных фрагментов (параметр *if* директивы **parallel**)

Теперь при наличии учебного примера можно пояснить назначение параметра *if* директивы **parallel**.

При разработке параллельных алгоритмов и программ важно понимать, что организация параллельных вычислений приводит к появлению некоторых дополнительных накладных затрат – в частности, в параллельной программе затрачивается время на создание потоков, их активизацию, приостановку при завершении параллельных фрагментов и т.п. Тем самым, для достижения положительного эффекта сокращение времени вычислений за счет параллельного выполнения должно, по крайней мере, превышать временные затраты на организацию параллелизма. Для оценки целесообразности распараллеливания можно использовать параметр *if* директивы **parallel**, задавая при его помощи условие создания параллельного фрагмента (если условие параметра *if* не выполняется, блок директивы **parallel** выполняется как обычный последовательный код). Так, например, в нашем учебном примере можно ввести условие, определяющее минимальный размер матрицы, при котором осуществляется распараллеливание вычислений – программный код примера в этом случае может выглядеть следующим образом:

```
#include <omp.h>
#define NMAX 1000
#define LIMIT 100
main () {
    int i, j, sum;
    float a[NMAX][NMAX];
    <инициализация данных>
#pragma omp parallel for shared(a) private(i,j,sum) if (NMAX>LIMIT)
{
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=0; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n", i, sum);
    } /* Завершение параллельного фрагмента */
}
```

Пример 5.6. Пример использования параметра *if* директивы **parallel**

В приведенном примере параллельный фрагмент создается только, если порядок матрицы превышает 100 (т.е. когда обеспечивается некоторый минимальный объем выполняемых вычислений).

#### 5.4. Управление данными для параллельно-выполняемых потоков

Как уже отмечалось ранее, потоки параллельной программы выполняются в общем адресном пространстве и, как результат, все данные (переменные) являются общедоступными для всех параллельно выполняемых потоков. Однако в ряде случаев необходимо наличие переменных, которые были бы локальными для потоков (например, для того, чтобы потоки не оказывали влияния друг на друга). И обратно – при одновременном доступе нескольких потоков к общим данным необходимо

обеспечивать условия взаимоисключений (даный аспект организации доступа к общим данным будет рассмотрен в следующем подразделе).

#### 5.4.1. Определение общих и локальных переменных

Параметры *shared* и *private* директивы **for** для управления доступа к переменным уже использовались в примере 5.2. Параметр *shared* определяет переменные, которые будут общими для всех потоков. Параметр *private* указывает переменные, для которых в каждом потоке будут созданы локальные копии – они будут доступны только внутри каждого потока в отдельности (значения локальных переменных потока недоступны для других потоков). Параметры *shared* и *private* могут повторяться в одной и той же директиве несколько раз, имена переменных должны быть уже ранее определены и не могут повторяться в списках параметров *shared* и *private*.

По умолчанию все переменные программы являются общими. Такое соглашение приводит к тому, что компилятор не может указать на ошибочные ситуации, когда программисты забывают описывать локальные переменные потоков в параметре *private* (отсутствие таких описаний приводит к тому, что переменные будут восприниматься как глобальные). Для выявления таких ошибок можно воспользоваться параметром *default* директивы **parallel** для изменения правила по умолчанию:

```
default ( shared | none )
```

Как можно видеть, при помощи этого параметра можно отменить действие правила по умолчанию (`default(none)`) или восстановить правило, что по умолчанию переменные программы являются общими (`default(shared)`).

Следует отметить, что начальные значения локальных переменных не определены, а конечные теряются при завершении потоков. Для инициализации можно использовать параметр *firstprivate* директивы **for**, по которому начальные значения локальных переменных будут устанавливаться в значения, которые существовали в переменных до момента создания локальных копий. Запоминание конечных значений обеспечивается при помощи параметра *lastprivate*, в соответствии с которым значения локальных переменных копируются из потока, выполнившего последнюю итерацию. Поясним сказанное на примере рис. 5.5. На рисунке показана переменная *sum*, которая определена как *lastprivate* в директиве **parallel for**. Для этой переменной создаются локальные копии в каждом потоке, при завершении параллельного участка программного кода значение локальной переменной потока, выполнившего последнюю итерацию цикла, переписывается в исходную переменную **sum**.

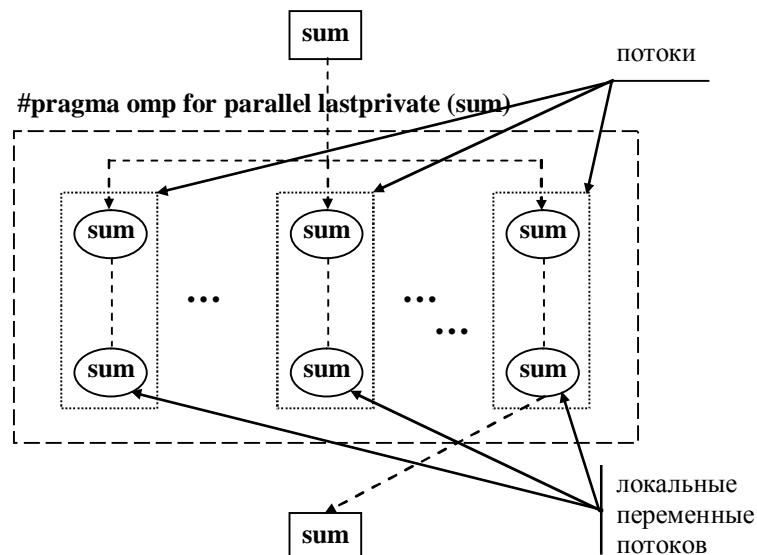


Рис. 5.5. Общая схема работы с локальными переменными потоков

#### 5.4.2. Совместная обработка локальных переменных (операция редукции)

Использование параметра *lastprivate* позволяет сохранить значений локальной переменной одного из потоков, но во многих случаях для обработки могут понадобиться значения всех локальных переменных. Данная возможность может быть обеспечена, например, сохранением этих значений в общих переменных – более подробно правила работы с общими переменными будет рассмотрена в следующем подразделе. Другой подход состоит в использовании коллективных операций над локальными переменными, предусмотренными в OpenMP. Задание коллективной операции происходит при помощи параметра *reduction* директивы **for**:

```
reduction (operator: list)
```

где список *list* задает набор локальных переменных (повторное описание в параметре *private* переменных из списка *list* не требуется), для которых должна быть выполнена коллективная операция, а поле *operator* указывает тип этой коллективной операции. Допустимыми значениями для поля *operator* являются следующие операции (которые не могут быть перегружены):

```
+, -, *, &, |, ^, &&, ||
```

Операция редукции обычно применяется к переменным, для которых вычисления задаются в виде выражений следующего вида:

- $x = x \langle\text{operator}\rangle \langle\text{expression}\rangle$
- $x = \langle\text{expression}\rangle \langle\text{operator}\rangle x$  (за исключением операции вычитания)
- $x \langle\text{op}\rangle= \langle\text{expression}\rangle$
- $x++, ++x, x--, --x$

где *x* есть имя скалярной переменной, выражение *expression* не должно включать переменную *x*, возможные операции для поля *operator* совпадают с выше приведенным списком, а допустимыми значениями для поля *op* являются операции:

```
+, -, *, &, |, ^
```

В качестве примера можно добавить в нашу учебную задачу действие по сложению всех сумм элементов строк матрицы – возможный программный код может быть следующим:

```
total = 0;
#pragma omp parallel for shared(a) private(i,j,sum) reduction (+:total)
{
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=i; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n", i, sum);
        total = total + sum;
    } /* Завершение параллельного фрагмента */
printf ("Общая сумма элементов матрицы равна %f\n", total);
```

Пример 5.7. Пример использования операции редукции данных

В качестве пояснений добавим, что по параметру *reduction* для переменной *total* создаются локальные переменные, затем полученные значения в потоках суммируются и запоминаются в исходной переменной. Подчеркнем, использование общей переменной *total* без создания локальных копий для накопления общей суммы при использовании в потоках операции

```
total = total + sum;
```

является неправильным, поскольку без обеспечения взаимоисключения при использовании общей переменной возникает ситуация гонки потоков и итоговый результат может быть ошибочным (см. следующий подраздел).

## 5.5. Организация взаимоисключения при использовании общих переменных

Как уже неоднократно отмечалось ранее, при изменении общих данных несколькими потоками должны быть обеспечены условия взаимоисключения – изменение значений общих переменных должно осуществляться в каждый конкретный момент времени только одним потоком. Рассмотрим возможные способы организации взаимоисключения.

### 5.5.1. Обеспечение атомарности (неделимости) операций

Действие над общей переменной может быть выполнено как атомарная (неделимая) операция при помощи директивы **atomic**. Формат директивы имеет вид:

```
#pragma omp atomic  
    <expression>
```

где *expression* должно иметь вид:

```
x++; или ++x; или x--; или --x;
```

где *x* есть имя любой целой скалярной переменной.

Директива **atomic** может быть записана и в виде:

```
#pragma omp atomic  
    x <operator>= <expression>
```

где *x* есть имя скалярной переменной, выражение *expression* не должно включать переменную *x*, а допустимыми значениями для поля *operator* являются следующие операции (которые не могут быть перегружены):

```
+, *, -, /, &, |, ^, >>, <<
```

Как следует из названия, операция директивы **atomic** выполняется как неделимое действие над указанной общей переменной, и, как результат, никакие другие потоки не могут получить доступ к этой переменной в этот момент времени.

Применим рассмотренную директиву для нашей учебной задачи при вычислении общей суммы:

```
total = 0;  
#pragma omp parallel for shared(a) private(i,j,sum)  
{  
    for (i=0; i < NMAX; i++) {  
        sum = 0;  
        for (j=i; j < NMAX; j++)  
            sum += a[i][j];  
        printf ("Сумма элементов строки %d равна %f\n",i,sum);  
#pragma omp atomic  
        total = total + sum;  
    } /* Завершение параллельного фрагмента */  
printf ("Общая сумма элементов матрицы равна %f\n",total);
```

Пример 5.8.

Пример использования директивы **atomic**

Отметим, что в данном случае потоки работают непосредственно с общей переменной *total*.

Как можно видеть, директива **atomic** может быть применена только для простых выражений, но является наиболее эффективным средством организации взаимоисключения, поскольку многие из допустимых для директивы операций на самом деле выполняются как атомарные на аппаратном уровне. Тем не менее, следует отметить, что данный вариант программы в общем случае будет проигрывать варианту 5.6 по эффективности, поскольку теперь синхронизация потоков будет выполняться для каждой строки обрабатываемой матрицы, а в примере 5.6 количество моментов синхронизации ограничено числом потоков.

### 5.5.2. Использование критических секций

Действия над общими переменными могут быть организованы в виде *критической секции*, т.е. как блок программного кода, который может выполняться только одним потоком в каждый конкретный момент времени. При попытке входа в критическую секцию, которая уже исполняется одним из потоков используется, все другие потоки приостанавливаются (*блокируются*). Как только критическая секция освобождается, один из приостановленных потоков (если они имеются) активизируется для выполнения критической секции.

Определение критической секции в OpenMP осуществляется при помощи директивы **critical**, формат записи которой имеет вид:

```
#pragma omp critical [(name)]
<block>
```

Как можно заметить, критические секции могут быть именованными – можно рекомендовать активное использование данной возможности для разделения критических секций, т.к. это позволит уменьшить число блокировок процессов.

Покажем использование механизма критических секций на примере нахождения максимальной суммы элементов строк матрицы. Одна из возможных реализаций состоит в следующем:

```
smax = -DBL_MAX;
#pragma omp parallel for shared(a) private(i,j,sum)
{
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=i; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n",i,sum);
        if ( sum > smax )
#pragma omp critical
            if ( sum > smax )
                smax = sum;
    } /* Завершение параллельного фрагмента */
printf ("Максимальная сумма элементов строк матрицы равна %f\n",smax);
```

Пример 5.9.

Пример использования критических секций

Следует обратить внимание на реализацию проверки суммы элементов строки на максимум. Директиву **critical** можно записать до первого оператора *if*, однако это приведет к тому, что критическая секция будет задействована для каждой строки и это приведет к дополнительным блокировкам потоков. Лучший вариант – организовать критическую секцию только тогда, когда необходимо осуществить запись в общую переменную *smax* (т.е. когда сумма элементов строки превышает значение максимума). Отметим особо, что после входа в критическую секцию необходимо повторно проверить переменную *sum* на максимум, поскольку после первого оператора *if* и до входа в критическую секцию значение *smax* может быть изменено другими потоками.

Отсутствие второго оператора *if* приведет к появлению трудно-выявляемой ошибки, учет подобных моментов представляет определенную трудность параллельного программирования.

### 5.5.3. Применение переменных семафорного типа (замков)

В OpenMP поддерживается специальный тип данных *omp\_lock\_t*, который близок к классическому понятию семафоров. Для переменных этого типа определены функции библиотеки OpenMP:

- Инициализировать замок:

```
void omp_init_lock(omp_lock_t *lock);
```

- Установить замок:

```
void omp_set_lock (omp_lock_t &lock);
```

Если при установке замок был установлен ранее, то поток блокируется.

- Освободить замок:

```
void omp_unset_lock (omp_lock_t &lock);
```

После освобождения замка при наличии блокированных на этом замке потоков один из них активизируется и замок снова отмечается как закрытый.

- Установить замок без блокировки:

```
int omp_test_lock (omp_lock_t &lock);
```

Если замок свободен, функция его закрывает и возвращает значение *true*. Если замок занят, поток не блокируется, и функция возвращает значение *false*.

- Перевод замка в неинициализированное состояние:

```
void omp_destroy_lock(omp_lock_t &lock)
```

Переработаем пример 5.8 так, чтобы для организации взаимного исключения при доступе к общим данным использовался механизм замков:

```
omp_lock_t lock;
omp_init_lock(&lock);
smax = -DBL_MAX;
#pragma omp parallel for shared(a) private(i,j,sum)
{
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=i; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n",i,sum);
        if ( sum > smax ) {
            omp_set_lock (&lock);
            if ( sum > smax )
                smax = sum;
            omp_unset_lock (&lock);
        }
    } /* Завершение параллельного фрагмента */
printf ("Максимальная сумма элементов строк матрицы равна %f\n",smax);
omp_destroy_lock (&lock);
```

Пример 5.10. Пример организации взаимоисключения при помощи замков

В OpenMP поддерживаются также и вложенные замки, которые предназначены для использования в ситуациях, когда внутри критических секций осуществляется вызов одних и тех же замков. Механизм работы с вложенными замками является тем же

самым (в наименование функций добавляется поле *nest*), т.е. их использование обеспечивается при помощи функций:

```
void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t &lock);
void omp_unset_nest_lock (omp_nest_lock_t &lock);
int  omp_test_nest_lock (omp_nest_lock_t &lock);
void omp_destroy_nest_lock(omp_nest_lock_t &lock)
```

Как можно заметить, для описания вложенных замков используется тип *omp\_nest\_lock\_t*.

## 5.6. Распределение вычислительной нагрузки между потоками (распараллеливание по задачам при помощи директивы sections)

Напомним, что в рассмотренном ранее учебном материале для построения параллельных программ был изложен подход (см. подразделы 5.2 – 5.3), обычно именуемый *распараллеливанием по данным*, в рамках которого обеспечивается одновременное (параллельное) выполнение одних и тех же вычислительных действий над разными наборами обрабатываемых данных (например, как в нашем учебном примере, суммирование элементов разных строк матрицы). Другая также широко встречающаяся ситуация состоит в том, что для решения поставленной задачи необходимо выполнить разные процедуры обработки данных, при этом данные процедуры или полностью не зависят друг от друга, или же являются слабо связанными. В этом случае такие процедуры можно выполнить параллельно; такой подход обычно именуется *распараллеливанием по задачам*. Для поддержки такого способа организации параллельных вычислений в OpenMP для параллельного фрагмента программы, создаваемого при помощи директивы **parallel**, можно выделять параллельно выполняемые *программные секции* (директива **sections**).

Формат директивы **sections** имеет вид:

```
#pragma omp sections [<параметр> ...]
{
    #pragma omp section
    <блок_программы>
    #pragma omp section
    <блок_программы>
}
```

При помощи директивы **sections** выделяется программный код, который далее будет разделен на параллельно выполняемые секции. Директивы **section** определяют секции, которые могут быть выполнены параллельно (для первой по порядку секции директива **section** не является обязательной) – см. рис. 5.6. В зависимости от взаимного сочетания количества потоков и количества определяемых секций, каждый поток может выполнить одну или несколько секций (вместе с тем, при малом количестве секций некоторые потоки могут оказаться и без секций и окажутся незагруженными). Как результат, можно отметить, что использование секций достаточно сложно поддается масштабированию (настройке на число имеющихся потоков). Кроме того, важно помнить, что в соответствии со стандартом, порядок выполнения программных секций не определен, т.е. секции могут быть выполнены потоками в произвольном порядке.

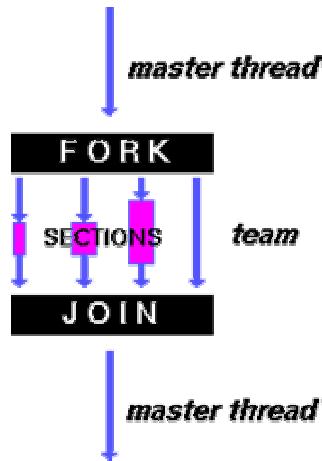


Рис. 5.6. Общая схема выполнения параллельных секций директивы **sections**

В качестве примера использования директивы расширим нашу учебную задачу действиями по копированию обрабатываемой матрицы (в качестве исходного варианта используется пример 5.6):

```

total = 0;
#pragma omp parallel shared(a,b) private(i,j)
{
#pragma omp sections
/* Вычисление сумм элементов строк и общей суммы */
for (i=0; i < NMAX; i++) {
    sum = 0;
    for (j=i; j < NMAX; j++)
        sum += a[i][j];
    printf ("Сумма элементов строки %d равна %f\n",i,sum);
    total = total + sum;
}
#pragma omp section
/* Копирование матрицы */
for (i=0; i < NMAX; i++) {
    for (j=i; j < NMAX; j++)
        b[i][j] = a[i][j];
}
/* Завершение параллельного фрагмента */
printf ("Общая сумма элементов матрицы равна %f\n",total);
  
```

### Пример 5.11. Пример использования директивы **sections**

Для обсуждения примера можно отметить, что выполняемые в программе вычисления (суммирование и копирование элементов) можно было бы объединить в рамках одних и тех же циклов, однако приведенное разделение тоже может оказаться полезным, поскольку в результате разделения данные вычисления могут быть легко векторизованы (конвейеризированы) компилятором. Важно отметить также, что сформированные подобным образом программные секции не сбалансированы по вычислительной нагрузке (первая секция содержит больший объем вычислений).

В качестве параметров директивы **sections** могут использоваться:

- `private (list)`
- `firstprivate (list)`
- `lastprivate (list)`
- `reduction (operator: list)`
- `nowait`

Все перечисленные параметры уже были рассмотрены ранее. Отметим, что по умолчанию выполнение директивы **sections** синхронизировано, т.е. потоки, завершившие свои вычисления, ожидают окончания работы всех потоков для одновременного завершения директивы.

В заключение можно добавить, что также как и для директивы **for**, в случае, если в блоке директивы **parallel** присутствует только директива **sections**, то данные директивы могут быть объединены. С использованием данной возможности пример 5.11 может быть переработан к виду:

```
total = 0;
#pragma omp parallel sections shared(a,b) private(i,j)
{
    /* Вычисление сумм элементов строк и общей суммы */
    for (i=0; i < NMAX; i++) {
        sum = 0;
        for (j=i; j < NMAX; j++)
            sum += a[i][j];
        printf ("Сумма элементов строки %d равна %f\n",i,sum);
        total = total + sum;
    }
#pragma omp section
    /* Копирование матрицы */
    for (i=0; i < NMAX; i++) {
        for (j=i; j < NMAX; j++)
            b[i][j] = a[i][j];
    }
} /* Завершение параллельного фрагмента */
printf ("Общая сумма элементов матрицы равна %f\n",total);
```

Пример 5.12.      Пример использования объединенной директивы  
**parallel sections**

## 5.7. Расширенные возможности OpenMP

### 5.7.1. Определение однопотоковых участков для параллельных фрагментов (директивы **single** и **master**)

При выполнении параллельных фрагментов может оказаться необходимым реализовать часть программного кода только одним потоком (например, открытие файла). Данную возможность в OpenMP обеспечивают директивы **single** и **master**.

Формат директивы **single** имеет вид:

```
#pragma omp single [<параметр> ...]
    <блок_программы>
```

Директива **single** определяет блок параллельного фрагмента, который должен быть выполнен только одним потоком; все остальные потоки ожидают завершения выполнения данного блока (если не указан параметр *nowait*) - см. рис. 5.7.

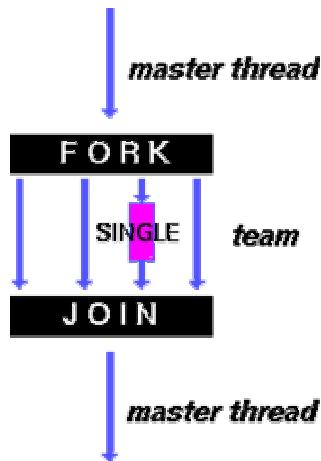


Рис. 5.7. Общая схема выполнения директивы **single**

В качестве параметров директивы могут использоваться:

- `private (list)`
- `firstprivate (list)`
- `copyprivate (list)`
- `nowait`

Новым в приведенном списке является только параметр `copyprivate`, который обеспечивает копирование переменных, указанных в списке *list*, после выполнения блока директивы **single** в локальные переменные всех остальных потоков.

Формат директивы **master** имеет вид:

```
#pragma omp master
<блок_программы>
```

Директива **master** определяет фрагмент кода, который должен быть выполнен только основным потоком; все остальные потоки пропускают данный фрагмент кода (завершение директивы по умолчанию не синхронизируется).

### 5.7.2. Выполнение барьерной синхронизации (директива **barrier**)

При помощи директивы **barrier** можно определить точку синхронизации, которую должны достигнуть все потоки для продолжения вычислений (директива может находиться в пределах как параллельного фрагмента, так и параллельной области, т.е. директива является отделяемой).

Формат директивы **barrier** имеет вид:

```
#pragma omp barrier
```

### 5.7.3. Синхронизация состояния памяти (директива **flush**)

Директива **flush** позволяет определить точку синхронизации, в которой системой должно быть обеспечено единое для всех потоков состояние памяти (т.е. если потоком какое-либо значение извлекалось из памяти для модификации, измененное значение обязательно должно быть записано в общую память).

Формат директивы **flush** имеет вид:

```
#pragma omp flush [(list)]
```

Как показывает формат, директива содержит список *list* с перечнем переменных, для которых выполняется синхронизация; при отсутствии списка синхронизация выполняется для всех переменных потока.

Следует отметить, что директива **flush** неявным образом присутствует в директивах **barrier**, **critical**, **ordered**, **parallel**, **for**, **sections**, **single**.

#### 5.7.4. Определение постоянных локальных переменных потоков (директива **threadprivate** и параметр **copyin** директивы **parallel**)

Как описывалось при рассмотрении директивы **parallel**, для потоков могут быть определены локальные переменные (при помощи параметров *private*, *firstprivate*, *lastprivate*), которые создаются в начале соответствующего параллельного фрагмента и удаляются при завершении потоков. В OpenMP имеется возможность создания и постоянно существующих локальных переменных для потоков при помощи директивы **threadprivate**.

Формат директивы **threadprivate** имеет вид:

```
#pragma omp threadprivate (list)
```

Список *list* содержит набор определяемых переменных. Созданные локальные копии не видимы в последовательных участках выполнения программы (т.е. вне параллельных фрагментов), но существуют в течение всего времени выполнения программы. Указываемые в списке переменные должны быть уже определены в программе; объявление переменных в директиве должно предшествовать использованию переменных в потоках.

Следует отметить, что использование директивы **threadprivate** позволяет решить еще одну проблему. Дело в том, что действие параметров *private* распространяется только на программный код параллельных фрагментов, но не параллельных областей – т.е., например, любая локальная переменная, определенная в параллельном фрагменте функции *root* на рис. 5.3, будет недоступна в параллельной области функции *node*. Выход из такой ситуации может быть или в передаче значений локальных переменных через параметры функций или же в использовании постоянных локальных переменных директивы **threadprivate**.

Отметим еще раз, что полученные в потоках значения постоянных переменных сохраняются между параллельными фрагментами программы. Значения этих переменных можно переустановить при начале параллельного фрагмента по значениям из основного потока при помощи параметра *copyin* директивы **parallel**.

Формат параметра *copyin* директивы **parallel** имеет вид:

```
copyin (list)
```

Для демонстрации рассмотренных понятий приведем пример A.32.1с из стандарта OpenMP 2.5.

```
#include <stdlib.h>
float* work;
int size;
float tol;
#pragma omp threadprivate(work,size,tol)
void a32( float t, int n )
{
    tol = t;
    size = n;
#pragma omp parallel copyin(tol,size)
    {
        build();
    }
}
void build()
{
```

```

    int i;
    work = (float*)malloc( sizeof(float)*size );
    for( i = 0; i < size; ++i ) work[i] = tol;
}

```

В приведенном примере определяются постоянно существующие локальные переменные *work*, *size*, *tol* для потоков (директива **threadprivate**). Перед началом параллельного фрагмента значения этих переменных из основного потока копируются во все потоки (параметр *copyin*). Значения постоянно существующих локальных переменных доступны в параллельной области функции *build* (для обычных локальных переменных потоков их значения пришлось бы передавать через параметры функции).

### 5.7.5. Управление количеством потоков

По умолчанию количество создаваемых потоков определяется реализацией и обычно совпадает с числом имеющихся вычислительных элементов в системе (процессоров и/или ядер)<sup>4)</sup>. При необходимости количество создаваемых потоков может быть изменено – для этой цели в OpenMP предусмотрено несколько способов действий.

Прежде всего, количество создаваемых потоков может быть задано при помощи параметра *num\_threads* директивы **parallel**.

Количество необходимых потоков может быть также задано при помощи функции **omp\_set\_num\_threads** библиотеки OpenMP.

Формат функции **omp\_set\_num\_threads** имеет вид:

```
void omp_set_num_threads (int num_threads);
```

Вызов функции **omp\_set\_num\_threads** должен осуществляться из последовательной части программы.

Для задания необходимого количества потоков можно воспользоваться и переменной окружения **OMP\_NUM\_THREADS**. При использовании нескольких способов задания наибольший приоритет имеет параметр *num\_threads* директивы **parallel**, затем функция библиотеки, затем переменная окружения.

Изменение количества создаваемых потоков может быть полезно и для целей отладки разрабатываемой параллельной программы для проверки ее работоспособности при разном количестве потоков.

Приведем здесь также дополнительные функции библиотеки OpenMP, которые могут быть использованы при работе с потоками:

- Получение максимально-возможного количества потоков:

```
int omp_get_max_threads(void)
```

- Получение фактического количества потоков в параллельной области программы:

```
int omp_get_num_threads(void)
```

- Получение номера потока:

```
int omp_get_thread_num(void)
```

- Получение числа вычислительных элементов (процессоров или ядер), доступных приложению:

```
int omp_get_num_procs(void)
```

<sup>4)</sup> Определение количества имеющихся вычислительных элементов осуществляется операционной системой. При этом следует понимать, что при аппаратной поддержке процессором технологии *многопоточности* (*hyperthreading*, *HT*) ОС будет воспринимать каждый процессор как несколько логических вычислительных элементов (по числу аппаратно поддерживаемых потоков).

### 5.7.6. Задание динамического режима при создании потоков

Количество создаваемых потоков в параллельных фрагментах программы по умолчанию является фиксированным (см. также предыдущий пункт), однако стандартом предусматривается возможность *динамического режима*, когда количество потоков может определяться реализацией для оптимизации функционирования вычислительной системы. Разрешение динамического режима и его отключение осуществляется при помощи функции **omp\_set\_dynamic** библиотеки OpenMP.

Формат функции **omp\_set\_dynamic** имеет вид:

```
void omp_set_dynamic (int dynamic);
```

Вызов функции **omp\_set\_dynamic** должен осуществляться из последовательной части программы. Функция разрешает (`dynamic=true`) или отключает (`dynamic=false`) динамический режим создания потоков.

Для управления динамическим режимом создания потоков можно воспользоваться и переменной окружения **OMP\_DYNAMIC**. При использовании нескольких способов задания наибольший приоритет имеет функция библиотеки, затем переменная окружения.

Для получения состояния динамического режима можно воспользоваться функцией **omp\_get\_dynamic** библиотеки OpenMP:

```
int omp_get_dynamic (void);
```

### 5.7.7. Управление вложенностью параллельных фрагментов

Параллельные фрагменты программы могут быть вложенными – такая возможность определяется реализацией OpenMP. По умолчанию для выполнения вложенных параллельных фрагментов создается столько же потоков, как и для параллельных фрагментов верхнего уровня.

Управление режимом выполнения вложенных фрагментов осуществляется при помощи функции **omp\_set\_nested** библиотеки OpenMP.

Формат функции **omp\_set\_nested** имеет вид:

```
void omp_set_nested (int nested);
```

Вызов функции **omp\_set\_nested** должен осуществляться из последовательной части программы. Функция разрешает (`nested=true`) или отключает (`nested=false`) режим поддержки вложенных параллельных фрагментов.

Для управления режимом поддержки вложенных параллельных фрагментов можно воспользоваться и переменной окружения **OMP\_NESTED**. При использовании нескольких способов задания наибольший приоритет имеет функция библиотеки, затем переменная окружения.

Для получения состояния режима поддержки вложенных параллельных фрагментов можно воспользоваться функцией **omp\_get\_nested** библиотеки OpenMP:

```
int omp_get_nested (void);
```

## 5.8. Дополнительные сведения

Итак, возможности технологии OpenMP рассмотрены практически полностью. В данном подразделе дополнительно рассматриваются правила разработки параллельных программ с использованием OpenMP на алгоритмическом языке Fortran, описывается возможность компиляции программы как обычного последовательного программного кода и приводится краткий перечень компиляторов, обеспечивающих поддержку OpenMP

### 5.8.1. Разработка параллельных программ с использованием OpenMP на алгоритмическом языке Fortran

Между разработкой параллельных программ с использованием OpenMP на алгоритмическом языке С и алгоритмических языках семейства Fortran (Fortran 77, Fortran 90 и Fortran 95) существует не так много различий.

Прежде всего формат директив OpenMP на алгоритмическом языке Fortran имеет следующий вид:

```
<маркер> <имя_директива> [<параметр>...]
```

Вид маркера зависит от формата записи программы:

- При фиксированном формате записи маркер имеет вид

```
!$omp или c$omp или *$omp
```

Маркер должен начинаться с позиции 1 строки. При этом сама директива может быть записана в несколько строк; первая строка директивы должна содержать в позиции 6 или пробел или 0; строки продолжения директивы в позиции 6 должны быть отмечены любым произвольным символом, отличающимся от пробела и 0.

- При свободном формате записи маркер должен иметь вид **!\$omp**. Маркер может быть записан, начиная с произвольной позиции строки, при этом маркеру должны предшествовать только пробелы или знаки табуляции. При размещении директивы на нескольких строках первая строка должна заканчиваться символом **&**, строки продолжения могут начинаться с этого знака, но приводить его в строках продолжения не обязательно.

В качестве основных отличий параллельных программ с использованием OpenMP на алгоритмическом языке Fortran отметим следующее:

1. Практически все директивы сопровождаются соответствующей директивой завершения; к директивам завершения относятся:

```
end critical  
end do  
end master  
end ordered  
end parallel  
end sections  
end single  
end workshare
```

2. В стандарте OpenMP для алгоритмического языка Fortran предусматривается дополнительная директива **workshare**, которая является близкой по своему назначению директиве **sections**.

3. Часть подпрограмм библиотеки OpenMP являются процедурами и, тем самым, должны вызываться при помощи оператора вызова процедуры **CALL**.

В качестве принятых соглашений при разработке программ на языке Fortran рекомендуется записывать имена подпрограмм с использованием прописных символов.

В качестве примера приведем вариант программы из примера 5.6 на алгоритмическом языке Fortran.

```
PROGRAM Example  
INCLUDE "omp_lib.h"  
INTEGER NMAX 1000  
INTEGER LIMIT 100  
INTEGER I, J, SUM
```

```

REAL A(NMAX,NMAX)
<инициализация данных>
!$OMP PARALLEL DO SHARED(A) PRIVATE(I,J,SUM) IF (NMAX>LIMIT)
    DO I = 1, NMAX
        SUM = 0
        DO J = 1, NMAX
            SUM = SUM + A(I,J)
        ENDDO
        PRINT * "Сумма элементов строки ", I, "равна ", SUM
    ENDDO
!$OMP END PARALLEL DO ! Завершение параллельного фрагмента
STOP
END

```

Пример 5.13. Пример параллельной программы с использованием OpenMP на алгоритмическом языке Fortran

### 5.8.2. Сохранение возможности последовательного выполнения программы

При разработке параллельной программы с использованием OpenMP в общем случае целесообразно сохранить возможность компиляции программы как обычного последовательного кода (обычно говорят о единственности программного кода для последовательного и параллельного вариантов программ). При соблюдении такого подхода значительно снижаются затраты на сопровождение и развитие программ – так, например, при необходимости изменений в общей части алгоритмов последовательных и параллельных вычислений не требуется проводить корректировку программного кода в разных вариантах программ. Указанный подход полезен и при использовании компиляторов, которые еще не поддерживают технологию OpenMP. И, наконец, последовательный вариант программы часто необходим для проведения вычислительных экспериментов при определении получаемого ускорения параллельных вычислений.

Поддержка единственности программного кода относится к числу важных положительных качеств технологии OpenMP – формат директив был специально подобран таким образом, чтобы "обычные" компиляторы без поддержки OpenMP воспринимали директивы как комментарии.

Однако проблемы поддержки единственности программного кода возникают при использовании в параллельной программе функций и переменных окружения OpenMP в явном виде. Данные проблемы могут быть решены с помощью директив условной компиляции средств препроцессирования. Стандартом OpenMP предусматривается, что компиляторы с поддержкой технологии OpenMP должны определять макропеременную `_OPENMP`, значением которой является дата поддерживаемого стандарта в формате ГГГГММ (год, месяц). Как результат, для обеспечения единственности программного кода последовательного и параллельного вариантов программ, все OpenMP-зависимые расширения параллельного варианта программы следует окружать директивами условной компиляции в виде:

```

#ifndef _OPENMP
    <OpenMP-зависимый программный код>
#endif

```

Последовательный вариант программы может порождаться и компиляторами с поддержкой OpenMP – для обеспечения такой возможности в составе OpenMP имеются функции-заглушки для всех функций библиотеки OpenMP. Указание подобного режима компиляции осуществляется при помощи соответствующих параметров компилятора.

### 5.8.3. Краткий перечень компиляторов с поддержкой OpenMP

Среди компиляторов с поддержкой OpenMP прежде всего следует отметить компиляторы компании Intel для алгоритмических языков Fortran и C/C++ для операционных систем семейств Unix и Windows. В следующей таблице приведены параметры этих компиляторов для управления возможностями OpenMP.

**Таблица.** Параметры компиляторов для управления возможностями OpenMP

Параметр		Описание
Windows	Linux	
-Qopenmp	-openmp	Компиляция программного кода с использованием OpenMP
-Qopenmp-profile	-openmp-profile	Компиляция с инструментацией программного кода с использованием OpenMP для обеспечения возможности применения Intel VTune Performance Analyzer для профилирования программы
-Qopenmp-stubs	-openmp-stubs	Компиляция программного кода как обычной последовательной программы (с игнорированием директив и использованием функций-заглушек для функций OpenMP)

Следует также помнить, что при сборке программы следует использовать библиотеки, обеспечивающие поддержки многопоточности.

Среди других компиляторов можно отметить:

- Fujitsu/Lahey Fortran, C and C++,
- HP Fortran, C and C++,
- IBM Fortran and C,
- Silicon Graphics. Fortran 77/90 (IRIX), C/C++,
- Portland Group (PGI). Fortran и C/C++ для Windows NT, Linux, Solaris (x86).

Полный перечень компиляторов, поддерживающих OpenMP представлен на портале [www.openmp.org](http://www.openmp.org).

## 5.9. Краткий обзор главы

Данная Глава посвящен рассмотрению методов параллельного программирования для вычислительных систем с общей памятью с использованием технологии OpenMP.

В самом начале Главы отмечается, что технология OpenMP является в настоящий момент времени одним из основных подходов для разработки параллельных программ для вычислительных систем с общей памятью (в т.ч. и для систем с активно развивающимися в последнее время *многоядерными процессорами*). В рамках данной технологии программист при разработке параллельной программы добавляет в программный код специальные директивы параллелизма для выделения в программе *параллельных фрагментов*, в которых последовательный исполняемый код может быть разделен на несколько раздельных командных *потоков (threads)*. Далее эти потоки могут исполняться на разных процессорах (процессорных ядрах) вычислительной системы.

В 5.1 рассматривается ряд понятий и определений, являющихся основополагающими для стандарта OpenMP. Так, дается представление *параллельной*

программы как набора последовательных (однопотоковых) и параллельных (многопотоковых) участков программного кода. Далее обсуждаются вопросы организации взаимодействия потоков с использованием общих данных и связанные с этим многие классические аспекты параллельного программирования – *гонка потоков, взаимное исключение, критические секции, синхронизация*. Приводится также структура и формат директив OpenMP.

В 5.2 проводится быстрое и простое введение в разработку параллельных программ с использованием OpenMP. Даётся описание директивы **parallel** и приводится пример первой параллельной программы с использованием OpenMP. В подразделе обсуждаются важные для дальнейшего рассмотрения понятия *фрагмента, области и секции* параллельной программы.

В 5.3 рассматриваются вопросы распределения вычислительной нагрузки между потоками на примере *распараллеливания циклов* по данным. Даётся описание директивы **for** и описываются способы управления распределением итераций цикла между потоками.

В 5.4 подробно обсуждаются вопросы управления данными для параллельно выполняемых потоков. Даётся понятие общих и локальных переменных для потоков. Приводится описание важной и часто встречающейся при обработке общих данных *операции редукции*.

В 5.5 рассматриваются вопросы организации взаимоисключения при использовании общих переменных. Среди описываемых подходов – использование *атомарных (неделимых) операций*, определение *критических секций*, применение семафоров специального типа (*замков*).

В 5.6 излагаются вопросы распределения вычислительной нагрузки между потоками на основе распараллеливания задач. Даётся описание директивы **sections**, приводятся примеры использования данной директивы.

В 5.7 рассматриваются расширенные возможности технологии OpenMP. Даётся описание ряда новых директив (**master, single, barrier, flush, threadprivate, copyin**), обсуждаются возможности управления потоками (количество создаваемых потоков, динамический режим создания потоков, вложенность параллельных фрагментов).

В 5.8 даются дополнительные сведения о технологии OpenMP – рассматриваются правила разработки параллельных программ с использованием OpenMP на алгоритмическом языке Fortran, описывается возможность компиляции программы как обычного последовательного программного кода и приводится краткий перечень компиляторов, обеспечивающих поддержку OpenMP.

## 5.10. Обзор литературы

В наиболее полном виде информация по параллельному программированию для вычислительных систем с общей памятью с использованием OpenMP содержится в [48]. Краткое описание OpenMP приводится в [8, 21], полезная информация представлена также в [1, 72, 85].

Достаточно много информации о технологии OpenMP содержится в сети Интернет. Так, могут быть рекомендованы информационно-аналитический портал [www.parallel.ru](http://www.parallel.ru) и, конечно же, ресурс [www.openmp.org](http://www.openmp.org).

Дополнительная информация по разработке многопоточных программ содержится в [7] (для ОС Windows) и [46] (стандарт POSIX Threads).

Для рассмотрения общих вопросов параллельного программирования для вычислительных систем с общей памятью может быть рекомендована работа [39].

## 5.11. Контрольные вопросы

1. Какие компьютерные платформы относятся к числу вычислительных систем с общей памятью?
2. Какие подходы используются для разработки параллельных программ?
3. В чем состоят основы технологии OpenMP?
4. В чем состоит важность стандартизации средств разработки параллельных программ?
5. В чем состоят основные преимущества технологии OpenMP?
6. Что понимается под параллельной программой в рамках технологии OpenMP?
7. Что понимается под понятием потока (thread)?
8. Какие возникают проблемы при использовании общих данных в параллельно выполняемых потоках?
9. Какой формат записи директив OpenMP?
10. В чем состоит назначение директивы **parallel**?
11. В чем состоит понятие фрагмента, области и секции параллельной программы?
12. Какой минимальный набор директив OpenMP позволяет начать разработку параллельных программ?
13. Как определить время выполнения OpenMP программы?
14. Как осуществляется распараллеливание циклов в OpenMP? Какие условия должны выполняться, чтобы циклы могли быть распараллелены?
15. Какие возможности имеются в OpenMP для управления распределением итераций циклов между потоками?
16. Как определяется порядок выполнения итераций в распараллеливаемых циклах в OpenMP?
17. Какие правила синхронизации вычислений в распараллеливаемых циклах в OpenMP?
18. Как можно ограничить распараллеливание фрагментов программного кода с невысокой вычислительной сложностью?
19. Как определяются общие и локальные переменные потоков?
20. Что понимается под операцией редукции?
21. Какие способы организации взаимоисключения могут быть использованы в OpenMP?
22. Что понимается под атомарной (неделимой) операцией?
23. Как определяется критическая секция?
24. Какие операции имеются в OpenMP для переменных семафорного типа (замков)?
25. В каких ситуациях следует применять барьерную синхронизацию?
26. Как осуществляется в OpenMP распараллеливание по задачам (директива **sections**)?
27. Как определяются однопотоковые участки параллельных фрагментов (директивы **single** и **master**)?
28. Как осуществляется синхронизация состояния памяти (директива **flush**)?
29. Как используются постоянные локальные переменные потоков (директивы **threadprivate** и **copyin**)?

30. Какие средства имеются в OpenMP для управления количеством создаваемых потоков?
31. Что понимается под динамическим режимом создания потоков?
32. Как осуществляется управление вложенностью параллельных фрагментов?
33. В чем состоят особенности разработки параллельных программ с использованием OpenMP на алгоритмическом языке Fortran?
34. Как обеспечивается единственность программного кода для последовательного и параллельного вариантов программы?
35. Какие компиляторы обеспечивают поддержку технологии OpenMP?

## Задачи и упражнения

1. Разработайте программу для нахождения минимального (максимального) значения среди элементов вектора.
2. Разработайте программу для вычисления скалярного произведения двух векторов.
3. Разработайте программу для задачи вычисления определенного интеграла с использованием метода прямоугольников

$$y = \int_a^b f(x)dx \approx h \sum_{i=0}^{N-1} f_i, \quad f_i = f(x_i), \quad x_i = i \cdot h, \quad h = (b - a) / N.$$

(описание методов интегрирования дано, например, в [68]).

4. Разработайте программу решения задачи поиска максимального значения среди минимальных элементов строк матрицы (такая задача имеет место для решения матричных игр)

$$y = \max_{1 \leq i \leq N} \min_{1 \leq j \leq N} a_{ij},$$

5. Разработайте программу для задачи 4 при использовании матриц специального типа (ленточных, треугольных и т.п.). Определите время выполнения программы и оцените получаемое ускорение. Выполните вычислительные эксперименты при разных правилах распределения итераций между потоками и сравните эффективность параллельных вычислений (выполнение таких экспериментов целесообразно выполнить для задач, в которых вычислительная трудоемкость итераций циклов является различной).

6. Реализуйте операцию редукции с использованием разных способов организации взаимоисключения (атомарные операции, критические секции, синхронизацию при помощи замков). Оцените эффективность разных подходов. Сравните полученные результаты с быстродействием операции редукции, выполняемой посредством параметра *reduction* директивы **for**.

7. Разработайте программу для вычисления скалярного произведения для последовательного набора векторов (исходные данные можно подготовить заранее в отдельном файле). Ввод векторов и вычисление их произведения следует организовать как две раздельные задачи, для распараллеливания которых используйте директиву **sections**.

8. Выполните вычислительные эксперименты с ранее разработанными программами при различном количестве потоков (меньше, равно или больше числа имеющихся вычислительных элементов). Определите время выполнения программ и оцените получаемое ускорение.

9. Уточните, поддерживает ли используемый Вами компилятор вложенные параллельные фрагменты. При наличии такой поддержки разработайте программы с использованием и без использования вложенного параллелизма. Выполните вычислительные эксперименты и оцените эффективность разных подходов.

10. Разработайте программу для задачи 4 с использованием распараллеливания циклов разного уровня вложенности. Выполните вычислительные эксперименты и сравните полученные результаты. Оцените величину накладных расходов на создание и завершение потоков.

## Приложение: Справочные сведения об OpenMP

### П1. Сводный перечень директив OpenMP

Для справки приведем перечень директив OpenMP с кратким пояснением их назначения.

**Таблица П1.** Сводный перечень директив OpenMP

Директива	Описание
<b>parallel</b> [<параметр>...]	Директива определения параллельного фрагмента в коде программы (подраздел 5.2). <b>Параметры:</b> <i>if, private, shared, default, firstprivate, reduction, copyin, num_threads</i>
<b>for</b> [<параметр>...]	Директива распараллеливания циклов (подраздел 5.2). <b>Параметры:</b> <i>private, firstprivate, lastprivate, reduction, ordered, nowait, schedule</i>
<b>sections</b> [<параметр>...]	Директива для выделения программного кода, который далее будет разделен на параллельно выполняемые параллельные секции (подраздел 5.6). Выделение параллельных секций осуществляется при помощи директивы <b>section</b> . <b>Параметры:</b> <i>private, firstprivate, lastprivate, reduction, nowait</i>
<b>section</b>	Директива выделения параллельных секций – должна располагаться в блоке директивы <b>sections</b> (подраздел 5.6).
<b>single</b> [<параметр>...]	Директива для выделения программного кода в параллельном фрагменте, исполняемого только одним потоком (п. 5.7.1). <b>Параметры:</b> <i>private, firstprivate, copyprivate, nowait</i>
<b>parallel for</b> [<параметр>...]	Объединенная форма директив <b>parallel</b> и <b>for</b> (подраздел 5.2). <b>Параметры:</b> <i>private, firstprivate, lastprivate, shared, default, reduction, ordered, schedule, copyin, if, num_threads</i>
<b>parallel sections</b> [<параметр>...]	Объединенная форма директив <b>parallel</b> и <b>sections</b> (подраздел 5.6). <b>Параметры:</b> <i>private, firstprivate, lastprivate,</i>

	<i>shared, default, reduction, copyin, if, num_threads</i>
<b>master</b>	Директива для выделения программного кода в параллельном фрагменте, исполняемого только основным ( <i>master</i> ) потоком (п. 5.7.1).
<b>critical [ (name) ]</b>	Директива для определения критических секций (п. 5.5.2).
<b>barrier</b>	Директива для барьерной синхронизации потоков (п. 5.7.2).
<b>atomic</b>	Директива для определения атомарной (неделимой) операции (п. 5.5.1).
<b>flush [list]</b>	Директива для синхронизации состояния памяти (п. 5.7.3).
<b>threadprivate (list)</b>	Директива для определения постоянных локальных переменных потоков (п. 5.7.4).
<b>ordered</b>	Директивы управления порядком вычислений в распараллеливаемом цикле (п. 5.3.2). При использовании данной директивы в директиве <b>for</b> должен быть указан одноименный параметр <b>ordered</b>

## П2. Сводный перечень параметров директив OpenMP

Для справки приведем перечень параметров директив OpenMP с кратким пояснением их назначения.

**Таблица П2.** Сводный перечень параметров директив OpenMP

Параметр	Описание
<b>private (list)</b>	Параметр для создания локальных копий для перечисленных в списке переменных для каждого имеющегося потока (п. 5.4.1). Исходные значения копий не определены. Директивы: <b>parallel, for, sections, single</b>
<b>firstprivate (list)</b>	Тоже что и параметр <b>private</b> и дополнительно инициализация создаваемых копий значениями, которые имели перечисленные в списке переменные перед началом параллельного фрагмента (п. 5.4.1). Директивы: <b>parallel, for, sections, single</b>
<b>lastprivate (list)</b>	Тоже что и параметр <b>private</b> и дополнительно запоминание значений локальных переменных после завершения параллельного фрагмента (п. 5.4.1). Директивы: <b>for, sections</b>
<b>shared (list)</b>	Параметр для определения общих переменных для всех имеющихся потоков (п. 5.4.1). Директивы: <b>parallel</b>
<b>default (shared   none)</b>	Параметр для установки правила по умолчанию на

	использование переменных в потоках (п. 5.4.1). Директивы: <b>parallel</b>
<b>reduction</b> ( <i>operator</i> : <i>list</i> )	Параметр для задания операции редукции (п. 5.4.2). Директивы: <b>parallel, for, sections</b>
<b>nowait</b>	Параметр для отмены синхронизации при завершении директивы. Директивы: <b>for, sections, single</b>
<b>if</b> ( <i>expression</i> )	Параметр для задания условия, только при выполнении которого осуществляется создание параллельного фрагмента (п. 5.3.4). Директивы: <b>parallel</b>
<b>ordered</b>	Параметр для задания порядка вычислений в распараллеливаемом цикле (п. 5.3.2). Директивы: <b>for</b>
<b>schedule</b> ( <i>type</i> [ , <i>chunk</i> ] )	Параметр для управления распределением итераций распараллеливаемого цикла между потоками (п. 5.3.1). Директивы: <b>for</b>
<b>copyin</b> ( <i>list</i> )	Параметр для инициализации постоянных переменных потоков (п. 5.7.4). Директивы: <b>parallel</b>
<b>copyprivate</b> ( <i>list</i> )	Копирование локальных переменных потоков после выполнения блока директивы <b>single</b> (п. 5.7.1). Директивы: <b>single</b>
<b>num_treads</b>	Параметр для задания количества создаваемых потоков в параллельной области (п. 5.7.5). Директивы: <b>parallel</b>

Приведем для наглядности сводную таблицу использования параметров в директивах OpenMP.

**Таблица П3.** Сводная таблица по использованию параметров в директивах OpenMP

Параметр	Директива					
	<b>parallel</b>	<b>for</b>	<b>sections</b>	<b>single</b>	<b>parallel for</b>	<b>parallel sections</b>
<b>if</b>	þ				þ	þ
<b>private</b>	þ	þ	þ	þ	þ	þ
<b>shared</b>	þ	þ			þ	þ
<b>default</b>	þ				þ	þ
<b>firstprivate</b>	þ	þ	þ	þ	þ	þ

<code>lastprivate</code>		þ	þ		þ	þ
<code>reduction</code>	þ	þ	þ		þ	þ
<code>copyin</code>	þ				þ	þ
<code>schedule</code>		þ			þ	
<code>ordered</code>		þ			þ	
<code>nowait</code>		þ	þ	þ		
<code>num_threads</code>	þ				þ	
<code>copyprivate</code>				þ		

### П3. Сводный перечень функций OpenMP

Для справки приведем перечень функций библиотеки OpenMP с кратким пояснением их назначения.

**Таблица П4.** Сводный перечень функций OpenMP

Функция	Описание
<code>void omp_set_num_threads (int num_threads)</code>	Установить количество создаваемых потоков (п. 5.7.5)
<code>int omp_get_max_threads (void)</code>	Получение максимально-возможного количества потоков (п. 5.7.5)
<code>int omp_get_num_threads (void)</code>	Получение количества потоков в параллельной области программы (п. 5.7.5)
<code>int omp_get_thread_num (void)</code>	Получение номера потока (п. 5.7.5)
<code>int omp_get_num_procs (void)</code>	Получение числа вычислительных элементов (процессоров или ядер), доступных приложению (п. 5.7.5)
<code>void omp_set_dynamic (int dynamic)</code>	Установить режим динамического создания потоков (п. 5.7.6)
<code>int omp_get_dynamic (void)</code>	Получение состояние динамического режима (п. 5.7.6)
<code>void omp_set_nested (int nested)</code>	Установить режим поддержки вложенных параллельных

	фрагментов (п. 5.7.7)
int <b>omp_get_nested</b> (void)	Получения состояния режима поддержки вложенных параллельных фрагментов (п. 5.7.7)
void <b>omp_init_lock</b> (omp_lock_t *lock) void <b>omp_init_nest_lock</b> (omp_nest_lock_t *lock)	Инициализировать замок (п. 5.5.3)
void <b>omp_set_lock</b> (omp_lock_t &lock) void <b>omp_set_nest_lock</b> (omp_nest_lock_t &lock)	Установить замок (п. 5.5.3)
void <b>omp_unset_lock</b> (omp_lock_t &lock) void <b>omp_unset_nest_lock</b> (omp_nest_lock_t &lock)	Освободить замок (п. 5.5.3)
int <b>omp_test_lock</b> (omp_lock_t &lock) int <b>omp_test_nest_lock</b> (omp_nest_lock_t &lock)	Установить замок без блокировки (п. 5.5.3)
void <b>omp_destroy_lock</b> (omp_lock_t &lock) void <b>omp_destroy_nest_lock</b> (omp_nest_lock_t &lock)	Перевод замка в неинициализированное состояние (п. 5.5.3)
double <b>omp_get_wtime</b> (void)	Получение времени текущего момента выполнения программы (п. 5.2.5)
double <b>omp_get_wtick</b> (void)	Получение времени в секундах между двумя последовательными показателями времени аппаратного таймера (п. 5.2.5)
int <b>omp_in_parallel</b> (void)	Проверка нахождения программы в параллельном фрагменте

#### П4. Сводный перечень переменных окружения OpenMP

Для справки приведем перечень переменных окружения OpenMP с кратким пояснением их назначения.

**Таблица П5.** Сводный перечень переменных окружения OpenMP

Переменная	Описание
<b>OMP_SCHEDULE</b>	Переменная для задания способа управления распределением итераций распараллеливаемого цикла между потоками (п. 5.3.1). <u>Значение по умолчанию:</u> <b>static</b>
<b>OMP_NUM_THREADS</b>	Переменная для задания количество потоков в параллельном фрагменте (п. 5.7.5). <u>Значение по умолчанию:</u> <b>количество вычислительных</b>

	<b>элементов (процессоров/ядер) в вычислительной системе.</b>
<b>OMP_DYNAMIC</b>	Переменная для задания динамического режима создания потоков (п. 5.7.6). <u>Значение по умолчанию: false.</u>
<b>OMP_NESTED</b>	Переменная для задания режима вложенности параллельных фрагментов (п. 5.7.7). <u>Значение по умолчанию: false.</u>

Глава 6. Параллельные методы умножения матрицы на вектор.....	1
6.1. Введение .....	1
6.2. Принципы распараллеливания .....	2
6.3. Постановка задачи .....	3
6.4. Последовательный алгоритм.....	4
6.5. Умножение матрицы на вектор при разделении данных по строкам .....	5
6.5.1. Выделение информационных зависимостей.....	5
6.5.2. Масштабирование и распределение подзадач по вычислительным элементам .....	6
6.5.3. Программная реализация .....	6
6.5.4. Анализ эффективности .....	7
6.5.5. Результаты вычислительных экспериментов .....	11
6.6. Умножение матрицы на вектор при разделении данных по столбцам .....	14
6.6.1. Определение подзадач и выделение информационных зависимостей .....	14
6.6.2. Масштабирование и распределение подзадач по вычислительным элементам .....	15
6.6.3. Программная реализация .....	15
6.6.4. Анализ эффективности .....	17
6.6.5. Результаты вычислительных экспериментов .....	20
6.7. Умножение матрицы на вектор при блочном разделении данных .....	23
6.7.1. Определение подзадач .....	23
6.7.2. Выделение информационных зависимостей.....	23
6.7.3. Масштабирование и распределение подзадач по вычислительным элементам .....	24
6.7.4. Программная реализация .....	25
6.7.5. Анализ эффективности .....	27
6.7.6. Результаты вычислительных экспериментов .....	29
6.8. Краткий обзор главы .....	33
6.9. Обзор литературы .....	34
6.10. Контрольные вопросы .....	34
6.11. Задачи и упражнения.....	35

## Глава 6. Параллельные методы умножения матрицы на вектор

### 6.1. Введение

Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

С учетом значимости эффективного выполнения матричных расчетов многие стандартные библиотеки программ содержат процедуры для различных матричных операций. Объем программного обеспечения для обработки матриц постоянно увеличивается – разрабатываются новые экономные структуры хранения для матриц специального типа (треугольных, ленточных, разреженных и т.п.), создаются различные высокоэффективные машинно-зависимые реализации алгоритмов, проводятся теоретические исследования для поиска более быстрых методов матричных вычислений.

Являясь вычислительно-трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений. С одной стороны, использование высокопроизводительных многопроцессорных систем позволяет существенно повысить сложность решаемых задач. С другой стороны, в силу своей достаточно простой формулировки матричные операции предоставляют прекрасную

возможность для демонстрации многих приемов и методов параллельного программирования.

В данном разделе обсуждаются методы параллельных вычислений для операции матрично-векторного умножения, в следующей Главе (Глава 7) рассматривается операция перемножения матриц. Важный вид матричных вычислений – решение систем линейных уравнений – представлен в Главе 8. Общий для всех перечисленных задач вопрос разделения обрабатываемых матриц между параллельно работающими потоками рассматривается в 6.2.

При изложении следующего материала будем полагать, что рассматриваемые матрицы являются *плотными* (*dense*), в которых число нулевых элементов является незначительным по сравнению с общим количеством элементов матриц.

## 6.2. Принципы распараллеливания

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между потоками. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

**1. Ленточное разбиение матрицы.** При *ленточном* (*block-striped*) разбиении каждому потоку выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы (рис. 6.1). Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе. При таком подходе для горизонтального разбиения по строкам, например, матрица  $A$  представляется в виде (см. рис. 6.1)

$$A = (A_0, A_1, \dots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \dots, a_{i_{k-1}}), i_j = ik + j, 0 \leq j < k, k = m/p, \quad (6.1)$$

где  $a_i = (a_{i1}, a_{i2}, \dots, a_{in})$ , где  $i < m$ , есть  $i$ -я строка матрицы  $A$  (предполагается, что количество строк  $m$  кратно числу вычислительных элементов (процессоров и/или ядер)  $p$ , т.е.  $m = kp$ ). Во всех алгоритмах матричного умножения и умножения матрицы на вектор, которые будут рассмотрены нами в этом и следующем разделах, используется разделение данных на непрерывной основе.

Другой возможный подход к формированию полос состоит в применении той или иной схемы *чертежования* (*цикличности*) строк или столбцов. Как правило, для чертежования используется число потоков  $p$  – в этом случае при горизонтальном разбиении матрица  $A$  принимает вид

$$A = (A_0, A_1, \dots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \dots, a_{i_{k-1}}), i_j = i + jp, 0 \leq j < k, k = m/p. \quad (6.2)$$

Циклическая схема формирования полос может оказаться полезной для лучшей балансировки вычислительной нагрузки вычислительных элементов (например, при решении системы линейных уравнений с использованием метода Гаусса – см. Главу 8).

**2. Блоchное разбиение матрицы.** При *блочном* (*chessboard block*) разделении матрица делится на прямоугольные наборы элементов – при этом, как правило, используется разделение на непрерывной основе. Пусть количество потоков составляет  $p = s \cdot q$ , количество строк матрицы является кратным  $s$ , а количество столбцов – кратным  $q$ , то

есть  $m = k \cdot s$  и  $n = l \cdot q$ . Представим исходную матрицу  $A$  в виде набора прямоугольных блоков следующим образом:

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots & A_{0q-1} \\ & \dots & & \\ A_{s-11} & A_{s-12} & \dots & A_{s-1q-1} \end{pmatrix},$$

где  $A_{ij}$  - блок матрицы, состоящий из элементов:

$$A_{ij} = \begin{pmatrix} a_{i_0j_0} & a_{i_0j_1} & \dots & a_{i_0j_{l-1}} \\ & \dots & & \\ a_{i_{k-1}j_0} & a_{i_{k-1}j_1} & \dots & a_{i_{k-1}j_{l-1}} \end{pmatrix}, i_v = ik + v, 0 \leq v < k, k = m/s, j_u = jl + u, 0 \leq u \leq l, l = n/q. \quad (6.3)$$

При таком подходе часто оказывается полезным, чтобы топология вычислительной системы имела (по крайней мере, на логическом уровне) вид решетки из  $s$  строк и  $q$  столбцов. В этом случае при разделении данных на непрерывной основе вычисления в большинстве случаев могут быть организованы таким образом, чтобы вычислительные элементы, соседние в структуре решетки, обрабатывали смежные блоки исходной матрицы. Следует отметить, что и для блочной схемы может быть применено циклическое чередование строк и столбцов.

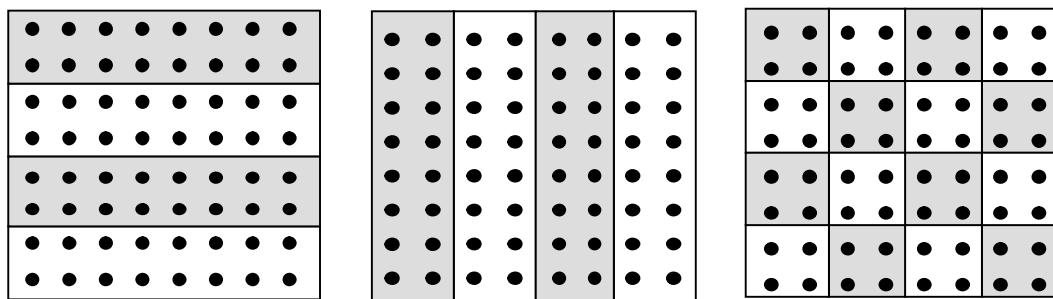


Рис. 6.1. Способы распределения элементов матрицы между потоками

В данной Главе рассматриваются три параллельных алгоритма для умножения квадратной матрицы на вектор. Каждый подход основан на разном типе распределения исходных данных (элементов матрицы и вектора) между потоками. Для каждого рассматриваемого алгоритма проводится теоретическая и экспериментальная оценка эффективности получаемых параллельных вычислений для определения наилучшего способа разделения данных.

### 6.3. Постановка задачи

В результате умножения матрицы  $A$  размерности  $m \times n$  и вектора  $b$ , состоящего из  $n$  элементов, получается вектор  $c$  размера  $m$ , каждый  $i$ -ый элемент которого есть результат скалярного умножения  $i$ -й строки матрицы  $A$  (обозначим эту строчку  $a_i$ ) и вектора  $b$ .

$$c_i = (a_i, b) = \sum_{j=1}^n a_{ij} b_j, 0 \leq i < m. \quad (6.4)$$

Тем самым, получение результирующего вектора  $c$  предполагает повторение  $m$  однотипных операций по умножению строк матрицы  $A$  и вектора  $b$ . Каждая такая операция включает умножение элементов строки матрицы и вектора  $b$  ( $n$  операций) и последующее суммирование полученных произведений ( $n-1$  операций). Общее количество необходимых скалярных операций есть величина

$$T_1 = m \cdot (2n - 1)$$

## 6.4. Последовательный алгоритм

Последовательный алгоритм умножения матрицы на вектор может быть представлен следующим образом.

```
// Алгоритм 6.1
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < m; i++) {
    c[i] = 0;
    for (j = 0; j < n; j++) {
        c[i] += A[i][j]*b[j]
    }
}
```

Алгоритм 6.1. Последовательный алгоритм умножения матрицы на вектор

Матрично-векторное умножение – это последовательность вычисления скалярных произведений. Поскольку каждое вычисление скалярного произведения векторов длины  $n$  требует выполнения  $n$  операций умножения и  $n-1$  операций сложения, его трудоемкость порядка  $O(n)$ . Для выполнения матрично-векторного умножения необходимо выполнить  $m$  операций вычисления скалярного произведения, таким образом, алгоритм имеет трудоемкость порядка  $O(mn)$ .

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем предполагать, что матрица  $A$  является квадратной, т.е.  $m=n$ .

Представим возможный вариант последовательной программы умножения матрицы на вектор.

**1. Главная функция программы.** Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 6.1
// Serial matrix-vector multiplication
void main(int argc, char* argv[]) {
    double* pMatrix; // Initial matrix
    double* pVector; // Initial vector
    double* pResult; // Result vector for matrix-vector multiplication
    int Size; // Sizes of initial matrix and vector

    // Data initialization
    ProcessInitialization(pMatrix, pVector, pResult, Size);

    // Matrix-vector multiplication
    SerialResultCalculation(pMatrix, pVector, pResult, Size);

    // Program termination
    ProcessTermination(pMatrix, pVector, pResult);
}
```

**2. Функция ProcessInitialization.** Эта функция определяет размер и элементы для матрицы  $A$  и вектора  $b$ . Значения для матрицы  $A$  и вектора  $b$  определяются в функции *RandomDataInitialization*.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
                           double* &pResult, int &Size) {
    int i; // Loop variable

    do {
        printf("\nEnter size of the initial objects: ");
        scanf("%d", &Size);
        if (Size <= 0) {
            printf("Size of the objects must be greater than 0! \n ");
        }
    } while (Size <= 0);
}
```

```

        }
    } while (Size <= 0);

pMatrix = new double [Size*Size];
pVector = new double [Size];
pResult = new double [Size];
for (i=0; i<Size; i++)
    pResult[i] = 0;
RandomDataInitialization(pMatrix, pVector, Size);
}

```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

**3. Функция SerialResultCalculation.** Данная функция производит умножение матрицы на вектор.

```

// Function for calculating matrix-vector multiplication
void SerialResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int i, j; // Loop variables
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}

```

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

Рассмотрим возможные способы организации параллельных вычислений для задачи умножения матрицы на вектор на многопроцессорных вычислительных системах с общей памятью. Как и ранее, для общности излагаемого учебного материала для упоминания одновременно и мультипроцессоров и многоядерных процессоров для обозначения одного вычислительного устройства будет использоваться понятие *вычислительного элемента* (*ВЭ*).

## 6.5. Умножение матрицы на вектор при разделении данных по строкам

Рассмотрим в качестве первого примера организации параллельных матричных вычислений алгоритм умножения матрицы на вектор, основанный на представлении матрицы непрерывными наборами (горизонтальными полосами) строк. При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция скалярного умножения одной строки матрицы на вектор. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата *c*.

### 6.5.1. Выделение информационных зависимостей

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений показана на рис. 6.2.

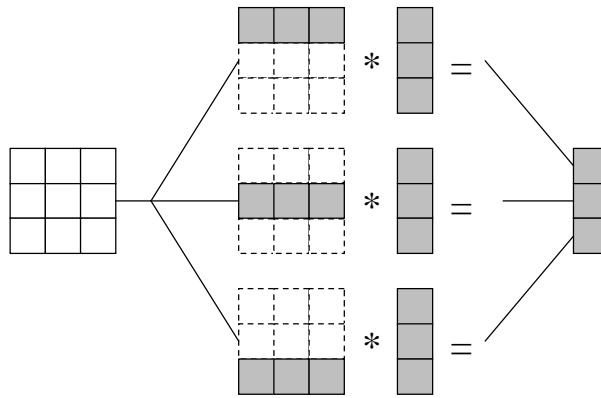


Рис. 6.2. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

### 6.5.2. Масштабирование и распределение подзадач по вычислительным элементам

В процессе умножения плотной матрицы на вектор количество вычислительных операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае, когда число вычислительных элементов  $p$  меньше числа базовых подзадач  $m$ , следует объединить базовые подзадачи таким образом, чтобы каждый ВЭ выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы  $A$ . В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов (блок) результирующего вектора  $c$ .

### 6.5.3. Программная реализация

Для того, чтобы разработать параллельную программу, реализующую описанный подход, при помощи технологии OpenMP, необходимо внести минимальные изменения в функцию умножения матрицы на вектор. Достаточно добавить одну директиву *parallel for* в функции *SerialResultCalculation* (назовем новый вариант функции *ParallelResultCalculation*):

```
// Function for parallel calculating matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
                                double* pResult, int Size) {
    int i, j; // Loop variables
#pragma omp parallel for private (j)
    for (i=0; i<Size; i++) {
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}
```

Данная функция производит умножение строк матрицы на вектор с использованием нескольких параллельных потоков. Каждый поток выполняет вычисления над несколькими соседними строками матрицы и, таким образом, получает блок результирующего вектора  $c$ .

Параллельные области в данной функции задаются директивой *parallel for*. Компилятор, поддерживающий технологию OpenMP, разделяет выполнение итераций цикла между несколькими потоками параллельной программы, количество которых обычно совпадает с числом вычислительных элементов (процессоров и/или ядер) в вычислительной системе. Параметр директивы *private* указывает необходимость создания отдельных копий для задаваемых переменных для каждого потока, данные копии могут использоваться в потоках независимо друг от друга.

В данной функции между потоками параллельной программы разделяются итерации внешнего цикла алгоритма умножения матрицы на вектор. В результате, каждый поток выполняет непрерывную последовательность итераций цикла по переменной  $i$ , и, таким образом, умножает горизонтальную полосу матрицы  $A$  на вектор  $b$  и вычисляет блок элементов результирующего вектора  $c$ .

#### 6.5.4. Анализ эффективности

Алгоритм умножения матрицы на вектор, основанный на ленточном горизонтальном разбиении матрицы, обладает хорошей «локализацией вычислений», то есть каждый поток параллельной программы использует только «свои» данные, и ему не требуются данные, которые в данный момент обрабатывает другой поток, нет обмена данными между потоками, не возникает необходимости синхронизации. Это означает, что в данной задаче практически нет накладных расходов на организацию параллелизма (за исключением расходов на организацию и закрытие параллельной секции), и можно ожидать линейного ускорения. Однако, как будет далее видно из представленных результатов (см. таблицу 6.3), ускорение, которое демонстрирует параллельный алгоритм умножения матрицы на вектор, основанный на ленточном горизонтальном разделении данных, далеко от линейного. В чем же причина такого поведения параллельной программы?

Задача умножения матрицы на вектор обладает сравнительно невысокой вычислительной сложностью – трудоемкость алгоритма имеет порядок  $O(n^2)$ . Такой же порядок -  $O(n^2)$  – имеет и объем данных, обрабатываемый алгоритмом умножения. Время решения задачи одним потоком складывается из времени, когда процессор непосредственно выполняет вычисления, и времени, которое тратится на чтение необходимых для вычислений данных из оперативной памяти в кэш память. При этом время, необходимое на чтение данных, может быть сопоставимо или даже превосходить время счета.

Проведем простой эксперимент: измерим время выполнения последовательной программы, которая суммирует все элементы матрицы, и сравним это время со временем выполнения умножения матрицы того же размера на вектор. Результаты вычислительных экспериментов приведены в таблице 6.1 и представлены на рис. 6.3.

**Таблица 6.1.** Сравнение времени выполнения алгоритма умножения матрицы на вектор и алгоритма суммирования всех элементов матрицы

Размер матрицы	Время выполнения умножения матрицы на вектор	Время выполнения суммирования элементов матрицы
1000	0,0076	0,0064
2000	0,0303	0,0260
3000	0,0687	0,0577
4000	0,1221	0,1023
5000	0,1909	0,1593
6000	0,2747	0,2288
7000	0,3741	0,3123
8000	0,4893	0,4082
9000	0,6186	0,5152
10000	0,7637	0,6364

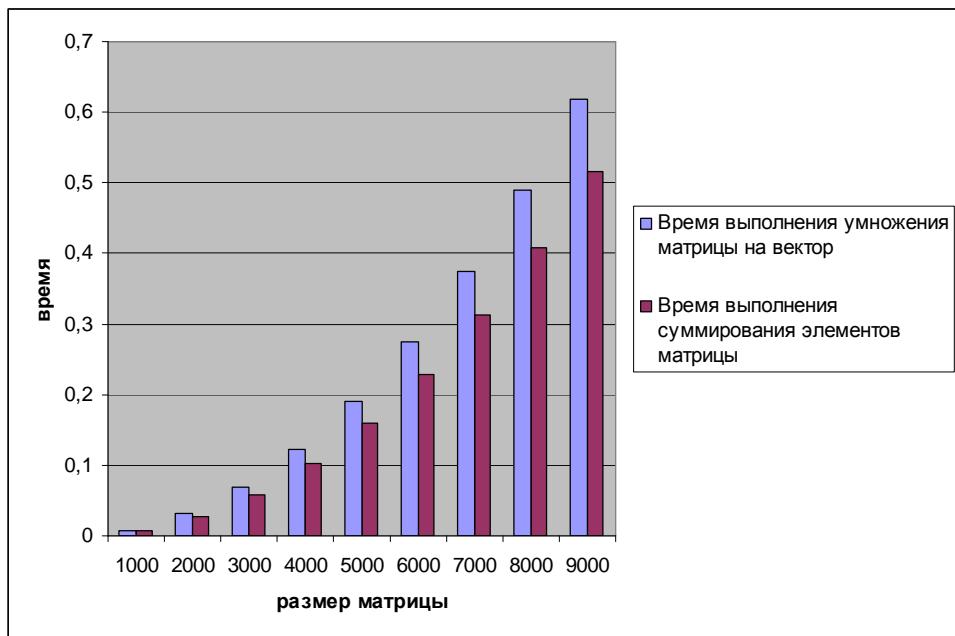


Рис. 6.3. Время выполнения умножения матрицы на вектор и суммирования всех элементов матрицы

При суммировании элементов матрицы на каждой итерации цикла выполняется простая операция сложения двух чисел. При умножении матрицы на вектор на каждой итерации цикла выполняются две операции: более сложная операция умножения и операция сложения. Но, несмотря на большую вычислительную сложность, время работы алгоритма умножения матрицы на вектор превосходит время выполнения сложения элементов матрицы всего на 16%. Этот эксперимент можно рассматривать, как подтверждение предположения о том, что значительная часть времени тратится на выборку необходимых данных из оперативной памяти в кэш процессора.

Процесс выполнения последовательного алгоритма умножения матрицы на вектор можно представить в виде диаграммы (рис. 6.4).

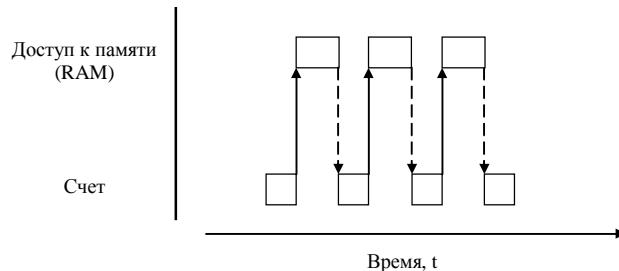


Рис. 6.4. Диаграмма состояний процесса выполнения последовательного алгоритма умножения матрицы на вектор

Следует отметить, что схема, представленная на рис. 6.4, является определенным упрощением процесса вычислений, реально выполняемых в компьютере, поскольку в современных процессорах на аппаратном уровне часто реализуются те или иные алгоритмы предсказания потребности данных для обработки и часть данных может перемещаться в кэш заранее (обеспечивая тем самым совмещение обработки и подкачки данных). Однако для предварительного анализа (тем более, для быстро выполняемых операций обработки) использование такой упрощенной схемы рис. 6.4. является вполне допустимой. Дополнительная информация по архитектуре современных процессоров может быть получена, например, в [35].

С учетом выполненного обсуждения, построим оценку времени выполнения матрично-векторного умножения. Итак, время выполнения последовательного алгоритма складывается из времени вычислений и времени доступа к памяти:

$$T_1 = T_{\text{calc}} + T_{\text{mem}} \quad (6.5)$$

Время выполнения вычислений можно определить как произведение количества выполненных операций  $N$  на время выполнения одной операции  $\tau$ :

$$T_{\text{calc}} = n \cdot (2n - 1) \cdot t,$$

где  $n$  – размерность матрицы (для получения более простых зависимостей предполагается, что все выполняемые вычислительные операции являются одинаковыми и имеют одну и ту же длительность  $t$ ).

Время доступа к памяти можно получить как результат деления объема извлекаемых из памяти данных  $M$  (в данном случае  $M=n^2$ ) на пропускную способность памяти<sup>1)</sup>  $\beta$ . Для точного построения модели следует учитывать, что данные загружаются в кэш память не побайтно, а целыми блоками – *кэши строками*. В вычислительной системе, на которой проводились все приведенные далее эксперименты, размер кэш строки был равен 64 байтам. Кроме того, при построении оценок необходимо принимать во внимание, что выполнению операций доступа к памяти может предшествовать некоторая задержка (*латентность доступа к памяти*)  $a$ .

В предельном случае, каждое обращение к данным приводит к чтению из памяти отдельной кэш строки и, как результат, время доступа к памяти можно оценить как:

$$T_{\text{mem}} = n^2 \cdot (a + 64/b) \quad (6.6)$$

Объединяя (6.5) и (6.6), получим суммарное время выполнения алгоритма:

$$T_1 = n \cdot (2n - 1) \cdot t + n^2 \cdot (a + 64/b) \quad (6.7)$$

Следует отметить, что построенная оценка рассчитана на наихудший случай, когда любое обращение к данным будет приводить к необходимости доступа к памяти. На самом деле, необходимые данные могут располагаться в кэш памяти и в этом случае необходимость доступа к оперативной памяти отпадает. Кэш память является существенно более быстрым видом памяти и использование данных из кэша приводит к заметному ускорению вычислений.

Потребность доступа к оперативной памяти возникает в моменты, когда необходимые данные отсутствуют в кэш памяти (такие ситуации носят наименование *кэши промахов*). Тем самым, фактические затраты на работу с памятью во многом определяются соотношением использования кэш памяти и более медленной оперативной памяти. Если данные в основном берутся из кэш памяти, затраты на доступ к памяти становятся незначительными (кэш память работает практически на частоте процессора). Если частота кэш промахов является большой, тогда затраты на доступ к памяти определяются быстродействием оперативной памяти.

Введем в разработанную ранее модель (6.7) величину  $g$ ,  $0 \leq g \leq 1$ , для задания *частоты возникновения кэши промахов*. Тогда оценка времени выполнения алгоритма матрично-векторного умножения принимает вид:

$$T_1 = n \cdot (2n - 1) \cdot t + g \cdot n^2 \cdot (a + 64/b) \quad (6.8)$$

Сведем воедино параметры построенной модели:

- $n$  – размерность матрицы,
- $a$  – латентность оперативной памяти,
- $\beta$  – пропускная способность оперативной памяти,

---

<sup>1)</sup> Пропускная способность памяти есть объем данных, который может быть извлечен из памяти за некоторую единицу времени

- $g$  – частота кэш промахов,
- $\tau$  – время выполнения базовой вычислительной операции.

Рассмотрим способы оценки значений этих параметров.

Параметр  $n$  является исходной величиной и задается при постановке задачи.

Следующие два параметра  $a$  и  $\beta$  являются характеристиками используемой вычислительной системы и для их измерения существует ряд систем анализа производительности памяти. В приводимых далее экспериментах использовалась система RightMark Memory Analyzer [88], с помощью которой для задействованной в экспериментах вычислительной системе были получены оценки  $\beta = 12,44 \text{ Гб/с}$  и  $a = 8,31 \text{ нс}$ .

Для оценки частота кэш промахов  $\gamma$  можно использовать систему Intel VTune [101]. Полезным средством для измерения данного параметра может оказаться и система Visual Performance System [24] (далее VPS), с помощью которой получались приводимые далее оценки.

Для того, чтобы оценить время одной операции  $\tau$ , можно измерить время выполнения последовательного алгоритма умножения матрицы на вектор при малых объемах данных, таких, чтобы матрица и вектор полностью поместились в кэш память. Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений можно заполнить матрицу и вектор случайными числами – выполнение этого действия гарантирует размещение данных в кэш памяти. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, можно получить время выполнения базовой вычислительной для алгоритма операции  $\tau$ . Для вычислительной системы, которая далее использовалась для проведения экспериментов, было получено значение  $\tau$ , равное 3,78 нс.

Получив оценки значений параметров, можно сравнить соответствие времен реального выполнения алгоритма матрично-векторного умножения и величин, получаемых с использованием разработанной модели (6.8)<sup>2)</sup>.

Частота кэш промахов, измеренная с помощью системы VPS, оказалась равной 0,0033.

**Таблица 6.2.** Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма умножения матрицы на вектор

Размер матриц	Эксперимент	Модель
1000	0,0076	0,0076
2000	0,0303	0,0304
3000	0,0687	0,0684
4000	0,1221	0,1216
5000	0,1909	0,1901
6000	0,2747	0,2737
7000	0,3741	0,3725
8000	0,4893	0,4866
9000	0,6186	0,6158
10000	0,7637	0,7603

<sup>2)</sup> Описание использованной вычислительной системы приведено в п. 6.5.5.

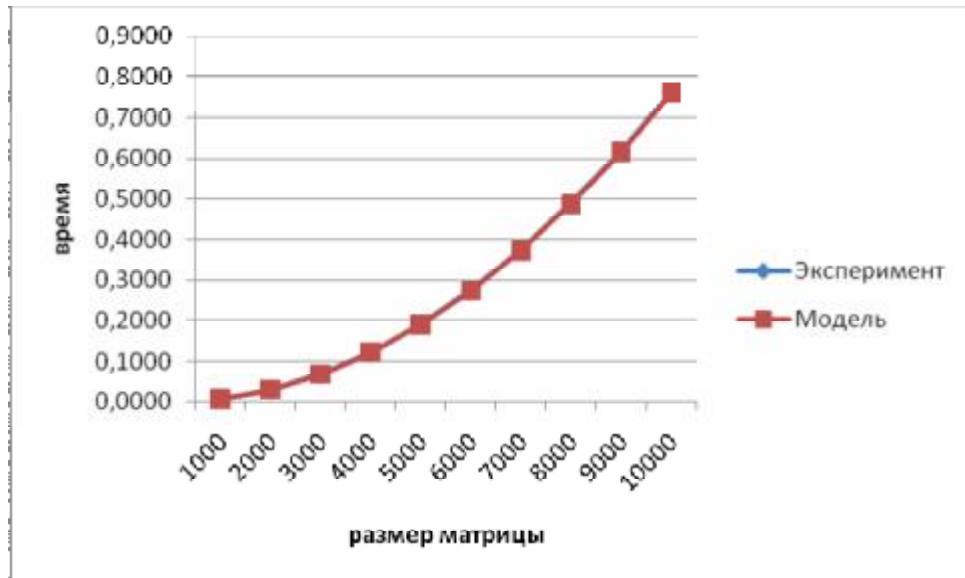


Рис. 6.5. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных (ленточное разбиение матрицы по строкам)

Как следует из приведенных результатов, разработанная модель в достаточно высокой степени позволяет оценить время выполнения последовательного алгоритма матрично-векторного умножения.

Рассмотрим теперь вопрос оценки времени выполнения параллельного алгоритма матрично-векторного умножения. В многоядерных процессорах Intel архитектуры Core 2 Duo ядра процессоров разделяют общий канал доступа к памяти. Тем самым, несмотря на то, что вычисления могут выполняться ядрами параллельно, доступ к памяти осуществляется строго последовательно. На рис. 6.6 представлена возможная диаграмма состояний потоков, выполняющих параллельный алгоритм умножения матрицы на вектор, при условии, что эти потоки запущены на ядрах одного двухъядерного процессора:

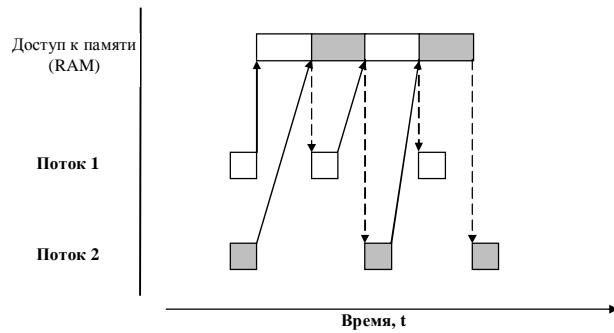


Рис. 6.6. Диаграмма состояний потоков при выполнении параллельного алгоритма умножения матрицы на вектор

Следовательно, время  $T_p$  выполнения параллельного алгоритма на компьютерной системе, имеющей  $p$  вычислительных элементов, с использованием  $p$  потоков и при описанной выше схеме доступа к памяти может быть оценено при помощи следующего соотношения:

$$T_p = \frac{n \cdot (2n-1)}{p} \cdot t + g \cdot n^2 \cdot (a + 64/b) \quad (6.9)$$

### 6.5.5. Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности параллельного алгоритма умножения матрицы на вектор. Эксперименты

проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

Результаты вычислительных экспериментов приведены в таблице 6.3 и графическом виде показаны на рис. 6.7. Времена выполнения алгоритмов указаны в секундах.

**Таблица 6.3.** Результаты вычислительных экспериментов для параллельного алгоритма умножения матрицы на вектор при ленточной схеме разделении данных по строкам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		Время	Ускорение	Время	Ускорение
1000	0,0076	0,0038	2,0224	0,0027	2,8148
2000	0,0303	0,0150	2,0233	0,0100	3,0315
3000	0,0688	0,0341	2,0167	0,0227	3,0301
4000	0,1222	0,0606	2,0163	0,0412	2,9654
5000	0,1909	0,0947	2,0164	0,0626	3,0500
6000	0,2748	0,1356	2,0263	0,0900	3,0531
7000	0,3741	0,1846	2,0266	0,1222	3,0615
8000	0,4894	0,2412	2,0286	0,1605	3,0491
9000	0,6186	0,3050	2,0286	0,2030	3,0475
10000	0,7637	0,3766	2,0281	0,2505	3,0483

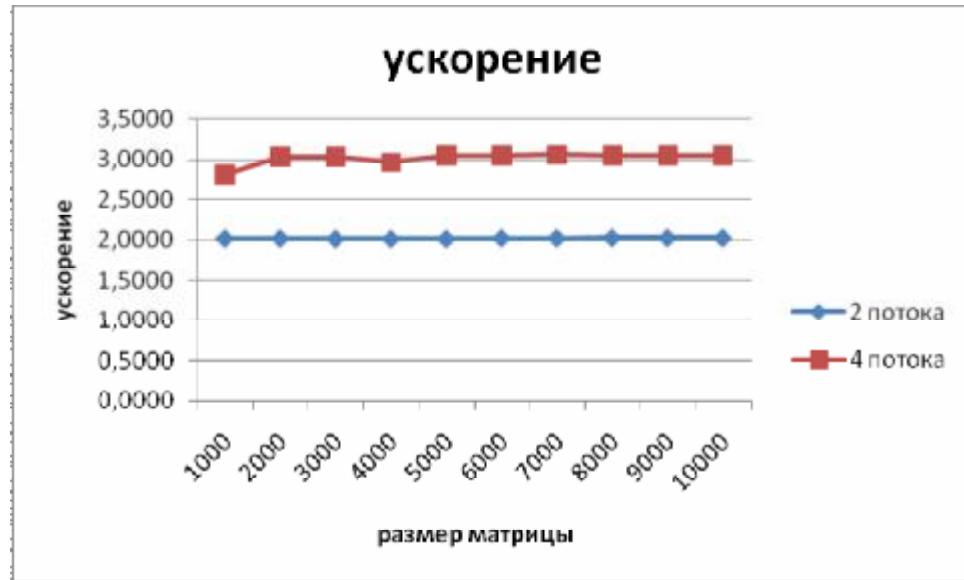


Рис. 6.7. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на ленточном горизонтальном разбиении матрицы

В таблицах 6.4 и 6.5 и на рис. 6.8 и 6.9 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матрицы на вектор с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (6.9). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0039, а для четырех потоков значение этой величины была оценена как 0,0143.

**Таблица 6.4.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на ленточном горизонтальном разбиении матрицы с использованием двух потоков

Размер матрицы	$T_p$	$T_p^* (calc)$ (модель)	Модель 6.6 – оценка сверху		Модель 6.9 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	0,0038	0,0038	0,0131	0,0169	0,0001	0,0038
2000	0,0150	0,0151	0,0524	0,0675	0,0002	0,0153
3000	0,0341	0,0340	0,1179	0,1519	0,0005	0,0345
4000	0,0606	0,0605	0,2096	0,2701	0,0008	0,0613
5000	0,0947	0,0945	0,3275	0,4220	0,0013	0,0958
6000	0,1356	0,1361	0,4716	0,6077	0,0018	0,1379
7000	0,1846	0,1852	0,6420	0,8272	0,0025	0,1877
8000	0,2412	0,2419	0,8385	1,0804	0,0033	0,2452
9000	0,3050	0,3062	1,0612	1,3674	0,0041	0,3103
10000	0,3766	0,3780	1,3101	1,6881	0,0051	0,3831

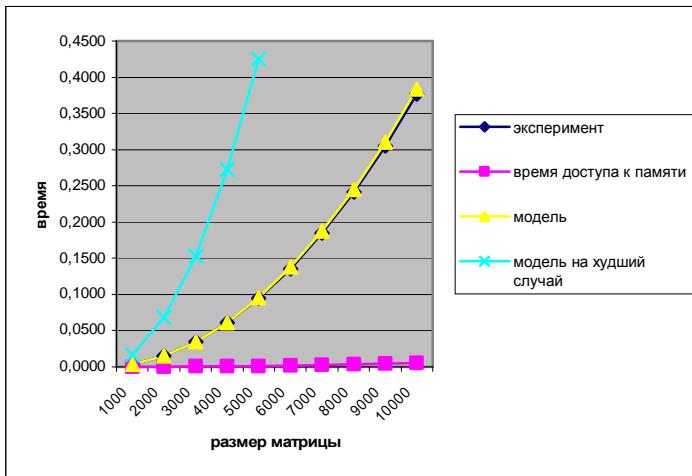


Рис. 6.8. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании двух потоков (ленточное разбиение матрицы по строкам)

**Таблица 6.5.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на ленточном горизонтальном разбиении матрицы с использованием четырех потоков

Размер матрицы	$T_p$	$T_p^* (calc)$ (модель)	Модель 6.6 – оценка сверху		Модель 6.9 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	0,0027	0,0019	0,0131	0,0150	0,0002	0,0021
2000	0,0100	0,0076	0,0524	0,0600	0,0007	0,0083
3000	0,0227	0,0170	0,1179	0,1349	0,0017	0,0187
4000	0,0412	0,0302	0,2096	0,2399	0,0030	0,0332
5000	0,0626	0,0472	0,3275	0,3748	0,0047	0,0519
6000	0,0900	0,0680	0,4716	0,5397	0,0067	0,0748
7000	0,1222	0,0926	0,6420	0,7346	0,0092	0,1018
8000	0,1605	0,1210	0,8385	0,9594	0,0120	0,1329

9000	0,2030	0,1531	1,0612	1,2143	0,0152	0,1683
10000	0,2505	0,1890	1,3101	1,4991	0,0187	0,2077



Рис. 6.9. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании четырех потоков (ленточное разбиение матрицы по строкам)

## 6.6. Умножение матрицы на вектор при разделении данных по столбцам

Рассмотрим теперь другой подход к параллельному умножению матрицы на вектор, основанный на разделении исходной матрицы на непрерывные вертикальные наборы (полосы) столбцов.

### 6.6.1. Определение подзадач и выделение информационных зависимостей

При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция умножения столбца матрицы  $A$  на один из элементов вектора  $b$ . Для организации вычислений в этом случае каждая базовая подзадача  $i$ ,  $0 \leq i < n$ , должна иметь доступ к  $i$ -ому столбцу матрицы  $A$ .

Каждая базовая задача  $i$  выполняет умножение своего столбца матрицы  $A$  на элемент  $b_i$ , в итоге в каждой подзадаче получается вектор  $c'(i)$  промежуточных результатов. Для получения элементов результирующего вектора  $c$  необходимо просуммировать вектора  $c'(i)$ , которые были получены каждой подзадачей.

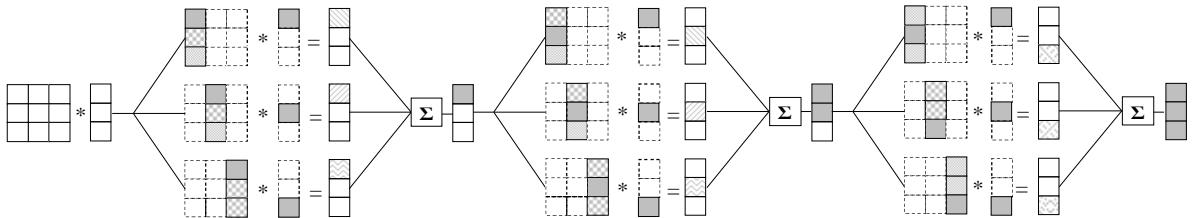


Рис. 6.10. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор с использованием разбиения матрицы по столбцам

Для того, чтобы реализовать такой подход, необходимо распараллелить внутренний цикл алгоритма умножения матрицы на вектор. В результате данного подхода параллельные области будут создаваться для вычисления каждого отдельного элемента

результатирующего вектора. Программа будет представляться в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков. Подобный принцип организации параллелизма получил название «*вилочного*» (*fork-join*) или *пульсирующего параллелизма*. При выполнении многопотокового участка каждый поток выполняет умножение одного элемента своего столбца исходной матрицы на один элемент вектора. Следующий за этим однопотоковый участок производит суммирование полученных результатов для того, чтобы вычислить элемент результирующего вектора (см. рис. 6.10). Таким образом, программа будет состоять из чередования  $n$  многопотоковых и  $n$  однопотоковых участков.

### 6.6.2. Масштабирование и распределение подзадач по вычислительным элементам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае, когда количество столбцов матрицы превышает число процессоров, базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних столбцов (в этом случае исходная матрица  $A$  разбивается на ряд вертикальных полос). При соблюдении равенства размера полос такой способ агрегации вычислений обеспечивает равномерность распределения вычислительной нагрузки по процессорам, составляющим многопроцессорную вычислительную систему.

### 6.6.3. Программная реализация

При реализации данного подхода к параллельному умножению матрицы на вектор каждый очередной многопотоковый участок параллельного алгоритма служит для вычисления одного элемента результирующего вектора. Очевидный способ реализовать подобный подход может состоять в следующем:

```
// Function for parallel calculating matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int i, j; // Loop variables
    for (i=0; i<Size; i++) {
#pragma omp parallel for
        for (j=0; j<Size; j++) // !!! Внимание - см. приводимые далее пояснения
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}
```

Выполним проверку правильности выполнения полученной программы – разработаем для этого функцию *TestResult*. Для контроля результатов выполним умножение исходной матрицы на вектор при помощи последовательного алгоритма, а затем сравним поэлементно вектора, полученные в результате выполнения последовательного и параллельного алгоритмов (следует отметить, что сравнение равенства вещественных чисел следует выполнять с некоторой явно указываемой точностью):

```
// Function for testing parallel matrix-vector multiplication
void TestResult(double* pMatrix, double* pVector, double* pResult, int Size) {
    double* pSerialResult = new double [Size];
    double Accuracy = 1.0e-6;
    int equal = 0;

    // Serial matrix-vector multiplication
    SerialResultCalculation(pMatrix, pVector, pSerialResult, Size);

    // Testing multiplication results
    for (int i=0; i<Size; i++)
        if (fabs(pResult[i]-pSerialResult[i])<=Accuracy) equal++;
```

```

    if (equal<Size>
        printf("The result is NOT correct \n");
    else
        printf ("The result is correct \n");

    delete [] pSerialResult;
}

```

Результатом работы этой функции является печать диагностического сообщения. Данная функция позволяет контролировать правильность выполнения параллельного алгоритма в автоматическом режиме, независимо от сложности и объема исходных данных.

Результаты экспериментов показывают, что разработанная функция параллельного умножения матрицы на вектор, реализующая разделение данных по столбцам, не всегда дает верный результат. Причина неправильной работы кроется в неправильном использовании переменных, являющихся общими для нескольких потоков. В разработанном варианте программы такими общими переменными являются элементы вектора *pResult*. В ходе вычислений несколько разных потоков могут, например, попытаться одновременно записать новые значения в одну и ту же общую переменную, что приведет к получению ошибочных результатов. Для исключения взаимовлияния потоки должны использовать общие данные с соблюдением правил взаимоисключений, которые бы обеспечивали использование общих переменных в каждый момент времени только одним потоком (а потоки, пытающиеся получить доступ к «занятым» общим данным должны блокироваться).

Реализация взаимоисключений доступа может состоять в использовании предусмотренных в OpenMP специальных переменных синхронизации - замков:

```

// Function for parallel calculating matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int i, j; // Loop variables
    omp_lock_t MyLock;
    omp_init_lock(&MyLock);

    for (i=0; i<Size; i++) {
#pragma omp parallel for
        for (j=0; j<Size; j++) {
            omp_set_lock(&MyLock);
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
            omp_unset_lock(&MyLock);
        }
    }
    omp_destroy_lock(&MyLock);
}

```

Функция *omp\_set\_lock* выполняется для потока только в том случае, если другие потоки не выполнили вызов этой функции для той же самой переменной; в противном случае поток блокируется (переменная-замок пропускает только один поток). Продолжение блокированных потоков осуществляется только после выполнения для замка функции *omp\_unset\_lock* (при наличии нескольких блокированных потоков после снятия замка выполнение разрешается только для одного потока и замок снова «закрывается»).

После внесения этих изменений в код функции, выполняющей умножение матрицы на вектор, результирующий вектор совпадает с вектором, полученным при помощи последовательного алгоритма. Но реализация алгоритма регулирования доступа к элементу вектора *pResult[i]* привела к тому, что потоки многопотокового участка могут выполняться только последовательно, так как один поток не может получить право на

изменение разделяемой переменной до тех пор, пока такое изменение выполняет другой поток.

Для одновременного решения проблемы разделения и защиты общих переменных в OpenMP реализован стандартный механизм эффективного выполнения операции редукции при распараллеливании циклов. Данный механизм состоит в использовании директивы *reduction*. Если при распараллеливании циклов указывается директива *reduction*, компилятор автоматически создает локальные копии переменной редуцирования для каждого потока параллельной программы, а после выполнения цикла собирает значения локальных копий в разделяемую переменную с использованием операции, указанной как аргумент директивы редукции.

Представим новый вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по столбцам. Ниже приведем только код основной функции, выполняющей параллельный алгоритм умножения матрицы на вектор.

```
// Function for parallel calculating matrix-vector multiplication
void ParallelResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int i, j; // Loop variables
    double IterGlobalSum = 0;

    for (i=0; i<Size; i++) {
        IterGlobalSum = 0;
#pragma omp parallel for reduction(+:IterGlobalSum)
        for (j=0; j<Size; j++)
            IterGlobalSum += pMatrix[i*Size+j]*pVector[j];
        pResult[i] = IterGlobalSum;
    }
}
```

#### 6.6.4. Анализ эффективности

При анализе эффективности данного параллельного алгоритма будем полагаться на результаты, полученные в подразделе 6.5.4. Очевидно, что ни время выполнения одной вычислительной операции, ни эффективная скорость доступа к оперативной памяти не зависят от того, каким образом распределяются вычисления между потоками параллельной программы.

Прежде всего, следует заметить, что при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении матрицы, выполняется больше арифметических операций. Действительно, на каждой итерации внешнего цикла после вычисления каждым потоком своей частичной суммы необходимо выполнить редукцию полученных результатов. Сложность выполнения операции редукции –  $\log_2 p$ . После выполнения редукции данных необходимо запомнить результат в очередной элемент результирующего вектора. Таким образом, время вычислений для данного алгоритма определяется формулой:

$$T_{calc} = \left( \frac{n \cdot (2n - 1)}{p} + n \cdot \log_2 p + n \right) \cdot t.$$

Как видно из представленного программного кода, на каждой итерации внешнего цикла алгоритма умножения матрицы на вектор организуется параллельная секция для вычисления каждого элемента результирующего вектора. Для организации и синхронизированного закрытия параллельных секций, а также для выполнения редукции вызываются специализированные функции библиотеки OpenMP. С одной стороны, для выполнения каждой такой функции необходимо определенное время. С другой стороны, в процессе выполнения эти функции используют данные, которые должны быть загружены в кэш процессора. Так как объем кэша ограничен, это ведет к вытеснению и последующей

повторной загрузке элементов матрицы и вектора. На выполнение таких подкачек данных тоже тратится некоторое время.

Для анализа программного кода и подтверждения выдвинутых предположений воспользуемся специализированной программной системой, предназначеннной для профилирования и оптимизации программ, Intel VTune Performance Analyzer (см., например, [107]). Данная программная система позволяет находить в программе критический (вычислительно-трудоемкий) код, измерять различные характеристики эффективности кода и наблюдать за процессом выполнения программы в динамике.

Проанализируем при помощи инструмента Call Graph (*Граф вызова функций*) профилировщика Intel VTune Performance Analyzer приложение, выполняющее параллельный алгоритм умножения матрицы на вектор, основанный на разделении матрицы на горизонтальные полосы, и сконцентрируем свое внимание на анализе выполнения функции *ParallelResultCalculation*. На рис. 6.11 представлен результат анализа. Видно, что собственное (без учета времени выполнения вызываемых функций) время и полное время выполнения функции совпадают. Это означает, что, несмотря на то, что функция *ParallelResultCalculation* вызывает другие функции (в данном случае – функции библиотеки времени исполнения OpenMP), практически все время тратится непосредственно на вычисления.

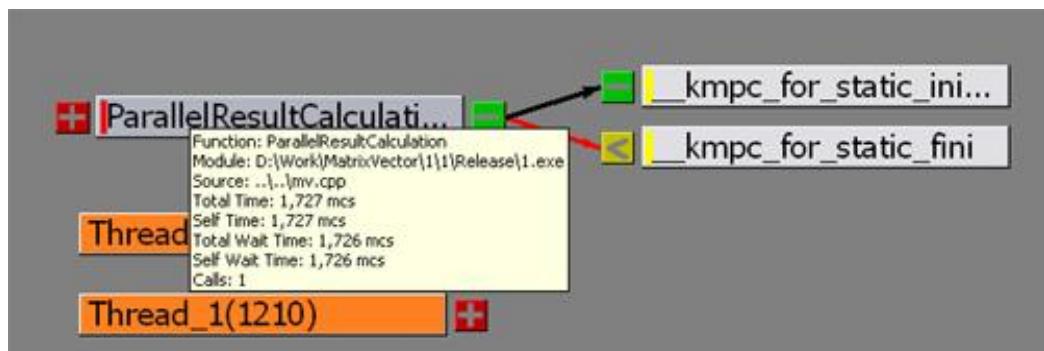


Рис. 6.11. Общее и собственное время выполнения функции *ParallelResultCalculation* в случае разделения матрицы на горизонтальные полосы

Теперь перейдем к анализу параллельного алгоритма, основанного на разделении матрицы по столбцам. В данном случае собственное время выполнения функции *ParallelResultCalculation* существенно меньше полного времени ее выполнения. Большая доля времени тратится на выполнение вложенных функций, которыми являются функции, необходимые для организации и закрытия параллельных секций, а также для выполнения редукции (рис. 6.12).

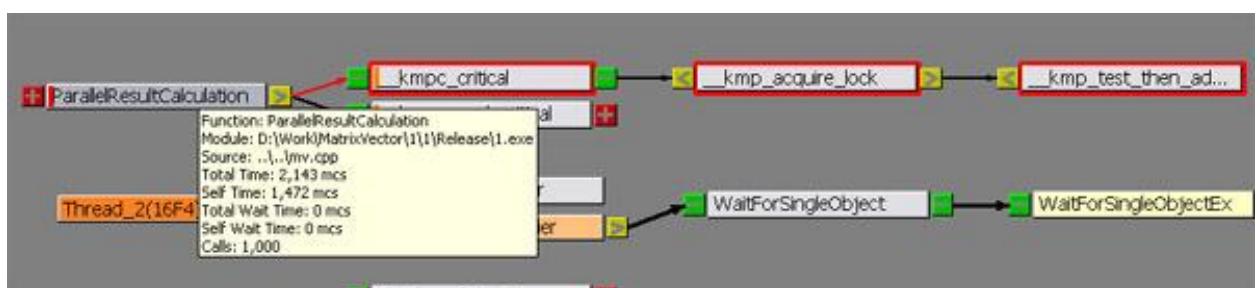


Рис. 6.12. Общее и собственное время выполнения функции *ParallelResultCalculation* в случае разделения матрицы на вертикальные полосы

Далее воспользуемся инструментом Sampling<sup>3)</sup> профилировщика Intel VTune Performance Analyzer, обеспечивающим автоматизированный процесс регистрации различных событий, возникающих в процессоре во время исполнения профилируемой программы. Создадим проект, в котором вызовем последовательно функцию, выполняющую параллельный алгоритм умножения матрицы на вектор, основанный на горизонтальном разделении матрицы (*ParallelResultCalculation1*), и функцию, выполняющую параллельный алгоритм, основанный на вертикальном разделении матрицы (*ParallelResultCalculation2*). Измерим число возникновений события выделения новой кэш-строки первого уровня (в Intel VTune Performance Analyzer данное событие называется L1 Cache Line Allocation). Результаты профилирования приложения представлены на рис. 6.13. Эксперименты проводились для матрицы размером  $1000 \times 1000$  элементов.

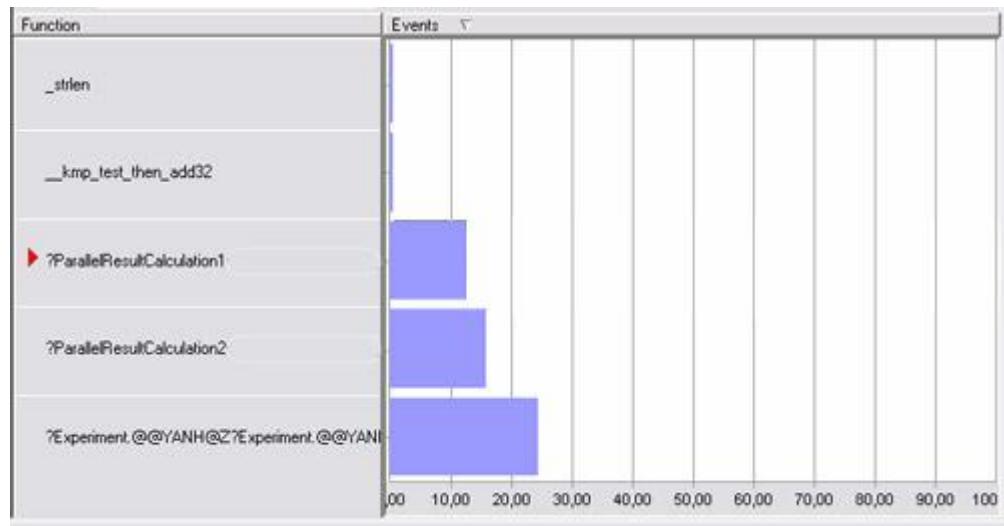


Рис. 6.13. Соотношение количества событий выделения кэш-линий первого уровня для двух параллельных алгоритмов умножения матрицы на вектор

При выполнении параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении, выделяется больше (в среднем на 25%) строк кэша первого уровня.

Далее, оценим накладные расходы на организацию параллельности на каждой итерации алгоритма, основанного на вертикальном разделении данных. Для этого подготовим еще один вариант параллельного алгоритма умножения матрицы на вектор при разделении матрицы по столбцам. Образуем в этом варианте программы потоки, которые самостоятельно определяют (используя идентификатор потока) блоки элементов матрицы и вектора для обработки. Далее потоки параллельно выполняют умножение своих столбцов матрицы на элементы вектора, результат умножения сохраняется в общем массиве *AllResults*. Количество элементов в этом массиве равно  $Size \times ThreadNum$ , где *ThreadNum* – общее число потоков. Потоки осуществляют запись частичных результатов в непересекающиеся сегменты массива *AllResults*: нулевой поток осуществляет запись в элементы с 0 по  $Size - 1$ , первый поток – в элементы с  $Size$  по  $2 \cdot Size - 1$ , и т.д. После выполнения умножения элементы вектора *AllResults* необходимо просуммировать для получения элементов результирующего вектора.

```
void ParallelResultCalculation(double* pMatrix, double* pVector,
                                double* pResult, double* AllResults, int Size) {
```

<sup>3)</sup> В научно-технической литературе практически не встречается общепринятого именования инструмента Sampling на русском языке и для его обозначения часто используют транслитерацию вида Сэмплирование.

```

int ThreadNum;
#pragma omp parallel shared (ThreadNum)
{
    ThreadNum = omp_get_num_threads();
    int ThreadID = omp_get_thread_num();
    int BlockSize = Size/ThreadNum;
    double IterResult;
    for (int i=0; i<Size; i++) {
        IterResult = 0;
        for (int j=0; j<BlockSize; j++)
            IterResult += pMatrix[i*Size+j+ThreadID*BlockSize] *
                pVector[j+ThreadID*BlockSize];
        AllResults[Size*ThreadID+i] = IterResult;
    }
}
for (int i=0; i<Size; i++)
    for (int j=0; j<ThreadNum; j++)
        pResult[i] += AllResults[j*Size+i];
}

```

Как видно из представленного программного кода, при выполнении данного алгоритма параллельная секция создается только один раз, и, следовательно, время, необходимое для выполнения функций библиотеки OpenMP, отвечающих за организацию и закрытие параллельных секций, пренебрежимо мало. Для того, чтобы оценить накладные расходы  $\delta$  на организацию параллельности на каждой итерации алгоритма, необходимо из времени выполнения исходного параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении данных, вычесть время выполнения вновь разработанного параллельного алгоритма и поделить полученную разницу на количество создаваемых параллельных секций. Эксперименты показывают, что величина  $\delta$  равна 0.25 мкс. Таким образом, время выполнения параллельного алгоритма умножения матрицы на вектор, основанного на вертикальном разделении матрицы, может быть вычислено для наихудшего случая (без использования кэш памяти) по формуле:

$$T_p = \left( \frac{n \cdot (2n-1)}{p} + n \cdot \log_2 p + n \right) \cdot t + n^2 \cdot \left( a + \frac{64}{b} \right) + n \cdot d. \quad (6.10)$$

При использовании кэш памяти с частотой кэш промахов  $g$  соотношение (6.10) принимает вид:

$$T_p = \left( \frac{n \cdot (2n-1)}{p} + n \cdot \log_2 p + n \right) \cdot t + g \cdot n^2 \cdot \left( a + \frac{64}{b} \right) + n \cdot d. \quad (6.11)$$

### 6.6.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матрицы на вектор при разбиении данных по столбцам проводились при условиях, указанных в п. 6.5.5. Результаты вычислительных экспериментов приведены в таблице 6.6.

**Таблица 6.6.** Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по столбцам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		Время	Ускорение	время	Ускорение
1000	0,0076	0,0067	1,1335	0,0064	1,1929

2000	0,0303	0,0219	1,3825	0,0165	1,8349
3000	0,0688	0,0452	1,5218	0,0316	2,1748
4000	0,1222	0,0769	1,5897	0,0503	2,4292
5000	0,1909	0,1160	1,6465	0,0712	2,6812
6000	0,2748	0,1624	1,6918	0,0979	2,8081
7000	0,3741	0,2186	1,7116	0,1340	2,7918
8000	0,4894	0,2819	1,7359	0,1641	2,9818
9000	0,6186	0,3539	1,7479	0,2036	3,0380
10000	0,7637	0,4330	1,7639	0,2479	3,0812

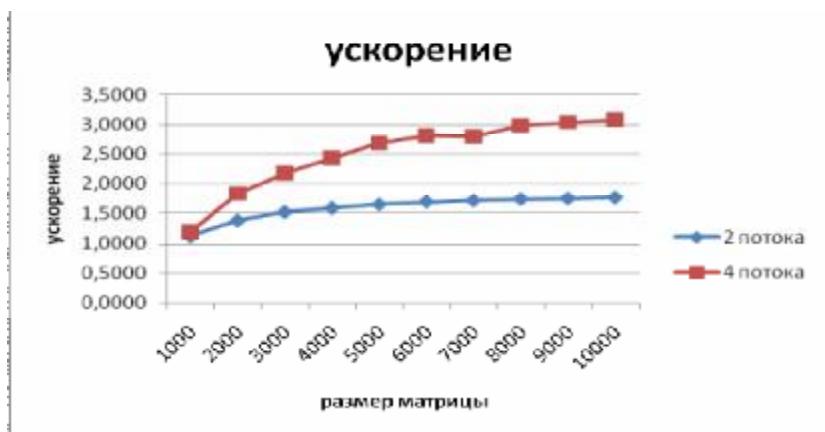


Рис. 6.14. Зависимость ускорения размера матрицы при выполнении параллельного алгоритма умножения матрицы на вектор (ленточное разбиение матрицы по столбцам)

В таблицах 6.7 и 6.8 и на рис. 6.15 и 6.16 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матрицы на вектор с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (6.11). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0039, а для четырех потоков значение этой величины была оценена как 0,0377.

**Таблица 6.7.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на ленточном разбиении матрицы по столбцам с использованием двух потоков

Размер матрицы	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 6.10 – оценка сверху		Модель 6.11 – уточненная оценка	
			$T_p^* \text{ (tem)}$	$T_p^*$	$T_p^* \text{ (tem)}$	$T_p^*$
1000	0,0067	0,0040	0,0131	0,0171	0,0001	0,0041
2000	0,0219	0,0156	0,0524	0,0680	0,0002	0,0158
3000	0,0452	0,0348	0,1179	0,1527	0,0005	0,0352
4000	0,0769	0,0615	0,2096	0,2711	0,0008	0,0623
5000	0,1160	0,0958	0,3275	0,4233	0,0013	0,0971
6000	0,1624	0,1376	0,4716	0,6093	0,0018	0,1395
7000	0,2186	0,1870	0,6420	0,8290	0,0025	0,1895
8000	0,2819	0,2440	0,8385	1,0825	0,0033	0,2472
9000	0,3539	0,3085	1,0612	1,3697	0,0041	0,3126
10000	0,4330	0,3806	1,3101	1,6907	0,0051	0,3857

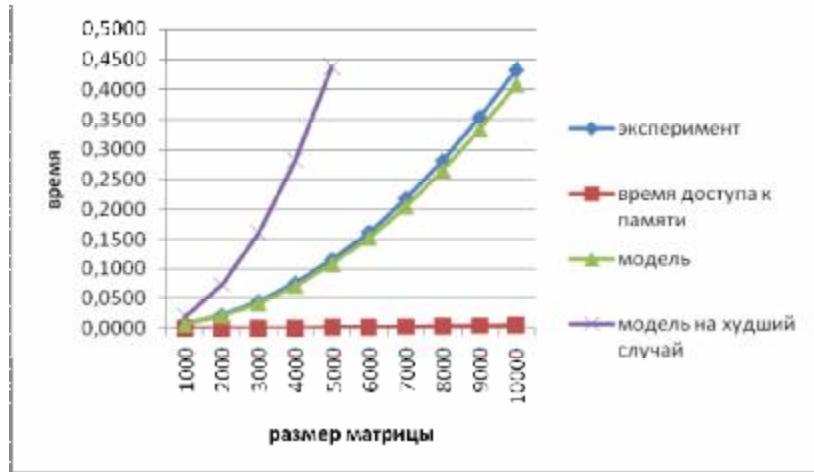


Рис. 6.15. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании двух потоков (ленточное разбиение матрицы по столбцам)

**Таблица 6.8.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на ленточном разбиении матрицы по столбцам с использованием четырех потоков

Размер матрицы	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 6.10 – оценка сверху		Модель 6.11 – уточненная оценка	
			$T_p^* \text{ (tem)}$	$T_p^*$	$T_p^* \text{ (tem)}$	$T_p^*$
1000	0,0064	0,0022	0,0131	0,0153	0,0005	0,0026
2000	0,0165	0,0081	0,0524	0,0605	0,0020	0,0101
3000	0,0316	0,0178	0,1179	0,1357	0,0044	0,0222
4000	0,0503	0,0313	0,2096	0,2409	0,0079	0,0392
5000	0,0712	0,0486	0,3275	0,3761	0,0123	0,0609
6000	0,0979	0,0696	0,4716	0,5413	0,0178	0,0874
7000	0,1340	0,0944	0,6420	0,7364	0,0242	0,1186
8000	0,1641	0,1230	0,8385	0,9615	0,0316	0,1547
9000	0,2036	0,1554	1,0612	1,2166	0,0400	0,1954
10000	0,2479	0,1916	1,3101	1,5017	0,0494	0,2410

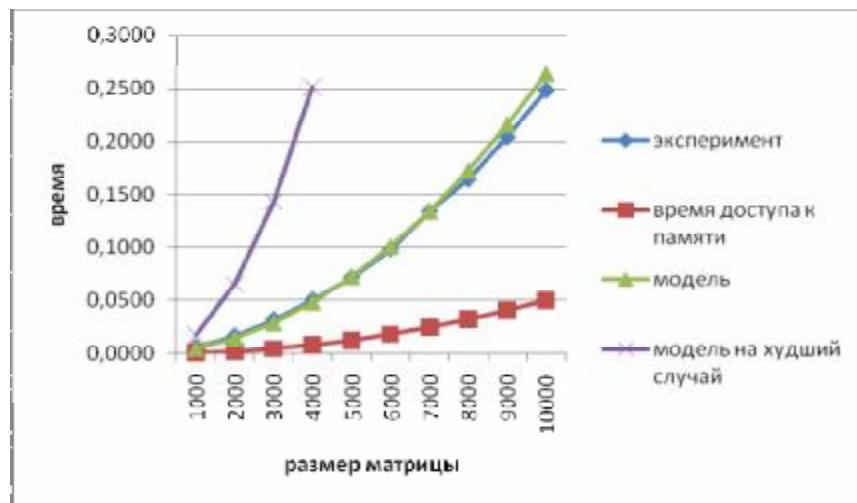


Рис. 6.16. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании четырех потоков (ленточное разбиение матрицы по столбцам)

## 6.7. Умножение матрицы на вектор при блочном разделении данных

Рассмотрим теперь параллельный алгоритм умножения матрицы на вектор, который основан на ином способе разделения данных – на разбиении матрицы на прямоугольные фрагменты (*блоки*).

### 6.7.1. Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 6.2. При таком способе разделения данных исходная матрица  $A$  представляется в виде набора прямоугольных блоков:

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots & A_{0q-1} \\ & \dots & & \\ A_{s-11} & A_{s-12} & \dots & A_{s-1q-1} \end{pmatrix},$$

где  $A_{ij}$ , где  $i < s$ ,  $j < q$ , есть блок матрицы:

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \dots & a_{i_0 j_{l-1}} \\ & \dots & & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & \dots & a_{i_{k-1} j_{l-1}} \end{pmatrix}, \quad i_v = ik + v, \quad 0 \leq v < k, \quad k = m/s, \quad j_u = jl + u, \quad 0 \leq u \leq l, \quad l = n/q$$

(здесь, как и ранее, предполагается, что  $p = s \cdot q$ , количество строк матрицы является кратным  $s$ , а количество столбцов – кратным  $q$ , то есть  $m = k \cdot s$  и  $n = l \cdot q$ ).

При использовании блочного представления матрицы  $A$  базовые подзадачи целесообразно определить на основе вычислений, выполняемых над матричными блоками. Для нумерации подзадач могут использоваться индексы располагаемых в подзадачах блоков матрицы  $A$ , т.е. подзадача  $(i,j)$  производит вычисления над блоком  $A_{ij}$ . Помимо блока матрицы  $A$  каждая подзадача должна иметь доступ к блоку вектора  $b$ . При этом для блоков одной и той же подзадачи должны соблюдаться определенные правила соответствия – операция умножения блока матрицы  $A_{ij}$  может быть выполнена только, если блок вектора  $b'(i,j)$  имеет вид

$$b'(i,j) = (b'_0(i,j), \mathbf{K}, b'_{l-1}(i,j)), \text{ где } b'_u(i,j) = b_{j_u}, \quad j_u = jl + u, \quad 0 \leq u < l, \quad l = n/q.$$

### 6.7.2. Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений для операции умножения матрицы на вектор при блочном разделении исходных данных. После перемножения блоков матрицы  $A$  и вектора  $b$  каждая подзадача  $(i,j)$  будет содержать вектор частичных результатов  $c'(i,j)$ , определяемый в соответствии с выражениями:

$$c'_n(i,j) = \sum_{u=0}^{l-1} a_{i_u j_u} b_{j_u}, \quad i_v = ik + v, \quad 0 \leq v < k, \quad k = m/s, \quad j_u = jl + u, \quad 0 \leq u \leq l, \quad l = n/q.$$

Как можно видеть из приведенных выражений, каждый поток вычисляет блок вектора частичных результатов исходной задачи умножения матрицы на вектор. Полный результат – вектор  $c$  – можно определить суммированием элементов блока вектора частичных результатов с одинаковыми индексами по всем подзадачам, относящимся к одним и тем же строкам решетки потоков, т.е.

$$c_n(i) = \sum_{j=0}^{q-1} c'_n(i, j), 0 \leq i < s, 0 \leq v < k, k = m/s.$$

Общая схема выполняемых вычислений для умножения матрицы на вектор при блочном разделении данных показана на рис. 6.17.

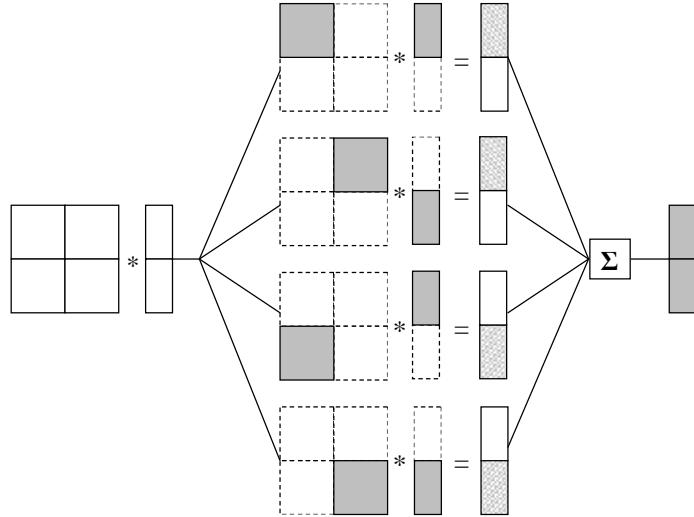


Рис. 6.17. Общая схема параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем полагать, что число блоков в матрице \$A\$ совпадает по горизонтали и по вертикали, т.е. \$s=q\$. Для эффективного выполнения параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении данных, целесообразно выделить число параллельных потоков совпадающим с количеством блоков матрицы \$A\$, т.е. такое количество потоков, которое является полным квадратом \$q^2\$. Данное требование может привести к тому, что число вычислительных элементов и количество потоков может различаться – для обозначения числа потоков в дальнейшем будем использовать переменную \$p\$, т.е. \$p=q^2\$. Дополнительно можно отметить заранее, что для эффективного выполнения вычислений количество потоков \$p\$ должно быть, по крайней мере, кратным числу вычислительных элементов \$r\$.

На рис. 6.17 приведена схема информационного взаимодействия подзадач в случае, когда для выполнения алгоритма создано 4 потока.

Рассмотрев представленную схему параллельных вычислений, можно сделать вывод о том, что информационная зависимость базовых подзадач проявляется только на этапе суммирования результатов перемножения блоков матрицы \$A\$ и блоков вектора \$b\$.

### 6.7.3. Масштабирование и распределение подзадач по вычислительным элементам

При сделанных предположениях общее количество базовых подзадач совпадает с числом выделенных потоков. Так, например, если определить число потоков \$p=q \times q\$, то размер блоков матрицы \$A\$ определяется соотношениями:

$$k=m/q, l=n/q,$$

где \$k\$ и \$l\$ есть количество строк и столбцов в блоках матрицы \$A\$. Такой способ определения размера блоков приводит к тому, что объем вычислений в каждой подзадаче является равным и, тем самым, достигается полная балансировка вычислительной нагрузки между потоками.

#### 6.7.4. Программная реализация

**1. Первый вариант.** Как уже отмечалось выше, для эффективного выполнения алгоритма умножения матрицы на вектор необходимо, чтобы количество параллельных потоков являлось полным квадратом. В приведенном ниже примере используется 4 потока (значение переменной *GridThreadsNum* устанавливается равным 4 – значение данной переменной должно переустанавливаться при изменении необходимого числа потоков). Определение числа потоков «вручную» до начала выполнения программы гарантирует корректную работу приложения на вычислительной системе с любым количеством доступных вычислительных элементов. Однако, если в вычислительной системе больше вычислительных элементов, чем определено параллельных потоков, эффективное использование ресурсов не может быть достигнуто.

Для определения количества параллельных потоков используется функция *omp\_set\_num\_threads* библиотеки OpenMP (функция должна быть вызвана в последовательном участке программы):

```
int omp_set_num_threads (int NumThreads);
```

Чтобы корректно определять блоки данных, над которыми поток должен выполнять вычисления, в программе необходимо иметь уникальный идентификатор потока. В качестве такого идентификатора может выступать номер потока. В библиотеке OpenMP существует функция для определения номера потока:

```
int omp_get_thread_num (void);
```

В представленном варианте алгоритма номер потока сохраняется в переменной *ThreadID*, которая при объявлении параллельной секции определена как *private*, то есть для этой переменной в каждом потоке создается локальная копия. Положения блоков матрицы и вектора, которые должны обрабатываться потоком, определяются в зависимости от значения переменной *ThreadID*.

Для накопления результатов умножения блоков матрицы и вектора в каждом потоке используется локальная область памяти *pThreadResult*. Для того, чтобы получить элемент результирующего вектора, необходимо просуммировать соответствующие элементы векторов *pThreadResult* со всех потоков. Для обеспечения контроля доступа к разделяемому ресурсу *pResult* со всех потоков параллельной программы используется механизм *критических секций*. Для объявления критической секции служит директива:

```
#pragma omp critical
    structured block
```

Блок операций, следующий за объявлением критической секции, в каждый момент времени может выполняться только одним потоком. Подобная организация программы гарантирует единственность доступа потоков для изменения разделяемых данных внутри критической секции.

Представим возможный вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по блокам. Ниже приведем только код основной функции, выполняющей параллельный алгоритм умножения матрицы на вектор.

```
// Function for parallel matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
    int Size) {
    int ThreadID;
    int GridThreadsNum = 4;
    int GridSize = int (sqrt(double(GridThreadsNum)));
    int BlockSize = Size/GridSize; // we assume that the matrix size is
                                    // divisible by the grid size
    omp_set_num_threads(GridThreadsNum);
```

```

#pragma omp parallel private (ThreadID)
{
    ThreadID = omp_get_thread_num();
    double * pThreadResult = new double [Size];
    for (int i=0; i<Size; i++) {
        pThreadResult[i] = 0;
    }
    int i_start = (int(ThreadID/GridSize))*BlockSize;
    int j_start = (ThreadID%GridSize)*BlockSize;
    double IterResult;
    for (int i=0; i< BlockSize; i++) {
        IterResult = 0;
        for (int j=0; j < BlockSize; j++)
            IterResult += pMatrix[(i+i_start)*Size+(j+j_start)] * pVector[j+i_start];
        pThreadResult[i+i_start] = IterResult;
    }
#pragma omp critical
    for (int i=0; i<Size; i++) {
        pResult[i] += pThreadResult[i];
    }
    delete [] pThreadResult;
} // pragma omp parallel
}

```

Следует отметить, что для запоминания результатов потоков в *pThreadResult* и их объединения в массиве *pResult* используется несколько "упрощенная" схема реализации с целью получения более простого варианта программного кода.

**2. Второй вариант.** При реализации первого варианта блочного параллельного алгоритма умножения матрицы на вектор необходимо «вручную» определить блоки данных и блоки вектора, над которыми каждый поток выполняет вычисления. В то же время, стандарт OpenMP предусматривает возможность организации *вложенных параллельных секций*. Это значит, что внутри одной параллельной секции можно объявить вторую параллельную секцию, которая разделит каждый из потоков внешней параллельной секции на несколько вложенных потоков.

Продемонстрируем описанный подход в случае, когда при объявлении каждой новой параллельной секции поток выполнения разделяется на два. При перемножении матрицы на вектор для внешнего цикла объявим внешнюю параллельную секцию. Потоки этой секции разделяют между собой строки матрицы – каждый поток в своих вычислениях использует горизонтальную полосу матрицы. Далее, используя механизм вложенного параллелизма, объявим внутреннюю параллельную секцию: потоки этой параллельной секции делят между собой столбцы полосы матрицы. Таким образом, будет реализовано блочное разделение данных. Схема выполнения данного параллельного алгоритма представлена на рис. 6.18.

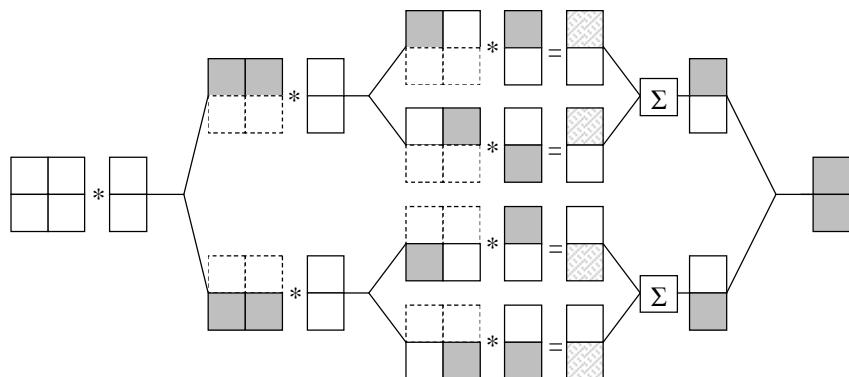


Рис. 6.18. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор с использованием разбиения матрицы на блоки и вложенного параллелизма

Использование механизма OpenMP для организации вложенного параллелизма позволяет значительно проще реализовать параллельный алгоритм умножения матрицы на вектор, основанный на блочном разделении матриц. Однако следует отметить, что на данный момент не все компиляторы, реализующие стандарт OpenMP, поддерживают вложенный параллелизм. Для компиляции представленного ниже кода использовался компилятор Intel C++ Compiler 10.0 for Windows, поддерживающий вложенный параллелизм.

Для того, чтобы включить поддержку вложенного параллелизма, необходимо вызвать функцию *omp\_set\_nested* библиотеки OpenMP (функция должна быть вызвана из последовательного участка программы):

```
void omp_set_nested(int nested);
```

Для разделения матрицы на полосы, необходимо разделить внешний цикл алгоритма умножения матрицы на вектор при помощи директивы *omp parallel for*. Для дальнейшего разделения полос матрицы на блоки – разделить внутренний цикл между потоками при помощи той же директивы. Поскольку для вычисления каждого элемента результирующего вектора создается параллельная секция, внутри которой каждый поток вычисляет частичную сумму для этого элемента, необходимо применить механизм редукции, подробно описанный в подразделе 6.6.

```
// Function for matrix-vector multiplication
void ResultCalculation(double* pMatrix, double* pVector, double* pResult,
    int Size) {
    int NestedThreadsNum = 2;
    omp_set_num_threads(NestedThreadsNum);
    omp_set_nested(true);
#pragma omp parallel for
    for (int i=0; i<Size; i++) {
        double ThreadResult = 0;
#pragma omp parallel for reduction(+:ThreadResult)
        for (int j=0; j<Size; j++)
            ThreadResult += pMatrix[i*Size+j]*pVector[j];
        pResult[i] = ThreadResult;
    }
}
```

Отметим, что в приведенной программе для задания количества потоков, создаваемых на каждом уровне вложенности параллельных областей, используется переменная *NestedThreadsNum* (в данном варианте программы ее значение устанавливается равным 2 – данное значение должно переустанавливаться при изменении необходимого числа потоков).

### 6.7.5. Анализ эффективности

**1. Первый вариант реализации.** При анализе эффективности первой реализации параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении матрицы, обратим внимание на дополнительные вычислительные операции: после того, как каждый поток выполнил умножение блока матрицы на блок вектора, необходимо сложить вектора частичных результатов для получения результирующего вектора. Если для выполнения параллельного алгоритма использовалось  $p$  потоков, то, с учетом использованной при реализации алгоритма схемы редукции данных, количество дополнительных операций равно  $n \cdot p$ . Таким образом, время вычислений можно определить по формуле:

$$T_{calc} = \left( \frac{n \cdot (2n-1)}{p} + n \cdot p \right) \cdot t .$$

Для оценки полного времени выполнения параллельного алгоритма можно использовать все положения, использованные при выводе необходимых аналитических соотношений для параллельного алгоритма при ленточном горизонтальном разделении данных (см. п. 6.5.4). Как результат, оценка сложности параллельных вычислений для наихудшего случая может быть представлена в следующем виде:

$$T_p = \left( \frac{n \cdot (2n-1)}{p} + n \cdot p \right) \cdot t + n^2 \cdot \left( a + \frac{64}{b} \right) . \quad (6.12)$$

При учете частоты кэш промахов соотношение (6.12) имеет вид:

$$T_p = \left( \frac{n \cdot (2n-1)}{p} + n \cdot p \right) \cdot t + g \cdot n^2 \cdot \left( a + \frac{64}{b} \right) . \quad (6.13)$$

Инструмент Call Graph профилировщика Intel VTune Performance Analyzer показывает, что затраты на организацию параллелизма не превосходят 5%.

**2. Второй вариант реализации.** Выполнив анализ второй вариант программной реализации параллельного алгоритма умножения матрицы на вектор, можно выделить дополнительные вычислительные операции: после вычисления частичного результата каждым параллельным потоком «внутренней» параллельной секции необходимо выполнить редукцию и записать полученный результат в соответствующий элемент результирующего вектора. Напомним, что для выполнения задачи используется  $p$  параллельных потоков,  $p=q \cdot q$ , и их число может отличаться от количества имеющихся вычислительных элементов (процессоров или ядер). С учетом данного обстоятельства можно определить, что время выполнения вычислений ограничено сверху величиной:

$$T_{calc} = \left( \frac{n \cdot (2n-1)}{p} + n \cdot q \right) \cdot t .$$

В случае же, когда количества вычислительных элементов совпадает с числом потоков, т.е.  $p=q$ , оценка времени вычислений может быть получена более точно:

$$T_{calc} = \left( \frac{n \cdot (2n-1)}{p} + \frac{n}{q} \cdot \log_2 q + \frac{n}{q} \right) \cdot t$$

При выполнении вычислений, «внутренние» параллельные секции создаются и закрываются много раз, кроме того, дополнительное время тратится на синхронизацию и выполнение операции редукции. Для оценки доли накладных расходов на обеспечение параллелизма снова воспользуемся профилировщиком Intel VTune Performance Analyzer (рис. 6.19).

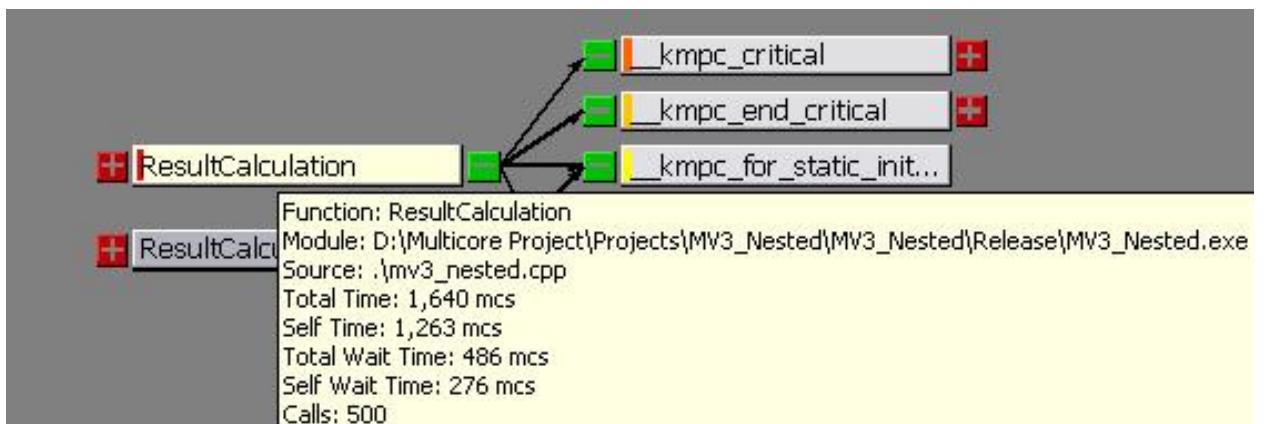


Рис. 6.19. Общее и собственное время выполнения функции *ParallelResultCalculation* при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на блочном разделении данных, реализованного с помощью вложенного параллелизма

Накладные расходы на организацию и закрытие параллельных секций были измерены при анализе параллельного алгоритма, основанного на вертикальном разделении данных (см. подраздел 6.6). В данном случае каждый поток создает параллельные секции  $n/q$  раз. Таким образом, время выполнения параллельного алгоритма может быть вычислено по формуле:

$$T_p = \left( \frac{n \cdot (2n - 1)}{p} + \frac{n}{q} \cdot \log_2 q + \frac{n}{q} \right) \cdot t + n^2 \cdot (a + \frac{64}{b}) + \frac{n}{q} \cdot d \quad (6.14)$$

Если же использовать информацию о частоте кэш промахов, то получим:

$$T_p = \left( \frac{n \cdot (2n - 1)}{p} + \frac{n}{q} \cdot \log_2 q + \frac{n}{q} \right) \cdot t + g \cdot n^2 \cdot (a + \frac{64}{b}) + \frac{n}{q} \cdot d \quad (6.15)$$

### 6.7.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные расчеты (см. п. 6.5.5).

**1. Первый вариант реализации.** Результаты экспериментов приведены в таблице 6.9 и на рис. 6.20.

**Таблица 6.9.** Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор при блочном разделении данных (четыре потока)

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	0,0076	0,0028	2,7066
2000	0,0303	0,0103	2,9403
3000	0,0688	0,0227	3,0276
4000	0,1222	0,0398	3,0697
5000	0,1909	0,0629	3,0347
6000	0,2748	0,0906	3,0338
7000	0,3741	0,1227	3,0496
8000	0,4894	0,1594	3,0697
9000	0,6186	0,2078	2,9764
10000	0,7637	0,2485	3,0730



Рис. 6.20. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма умножения матрицы на вектор (блочное разбиение матрицы, 4 потока)

В таблице 6.10 и на рис. 6.21 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матрицы на вектор с использованием четырех потоков со временем  $T_p^*$ , полученным при помощи модели (6.13). Частота кэш промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,015.

**Таблица 6.10.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на блочном разбиении матрицы с использованием четырех потоков

Размер матрицы	$T_p$	$T_p^* (calc)$ (модель)	Модель 6.12 – оценка сверху		Модель 6.13 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	0,0028	0,0019	0,0131	0,0150	0,0002	0,0021
2000	0,0103	0,0076	0,0524	0,0600	0,0008	0,0084
3000	0,0227	0,0170	0,1179	0,1349	0,0018	0,0188
4000	0,0398	0,0303	0,2096	0,2399	0,0031	0,0334
5000	0,0629	0,0473	0,3275	0,3748	0,0049	0,0522
6000	0,0906	0,0681	0,4716	0,5397	0,0071	0,0752
7000	0,1227	0,0927	0,6420	0,7346	0,0096	0,1023
8000	0,1594	0,1210	0,8385	0,9595	0,0126	0,1336
9000	0,2078	0,1531	1,0612	1,2144	0,0159	0,1691
10000	0,2485	0,1891	1,3101	1,4992	0,0197	0,2087

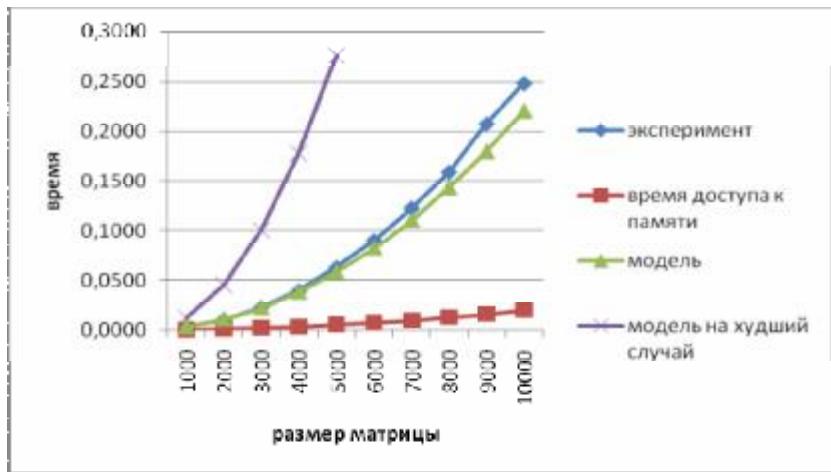


Рис. 6.21. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании четырех потоков (блочное разбиение матрицы)

**2. Второй вариант реализации.** Результаты экспериментов приведены в таблице 6.11. и на рис. 6.22.

**Таблица 6.11.** Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор при блочном разделении данных и использовании вложенного параллелизма

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	0,0076	0,0050	1,5173
2000	0,0303	0,0147	2,0629
3000	0,0688	0,0296	2,3272
4000	0,1222	0,0488	2,5021
5000	0,1909	0,0720	2,6520
6000	0,2748	0,1016	2,7035
7000	0,3741	0,1343	2,7852
8000	0,4894	0,1739	2,8150
9000	0,6186	0,2144	2,8857
10000	0,7637	0,2632	2,9013



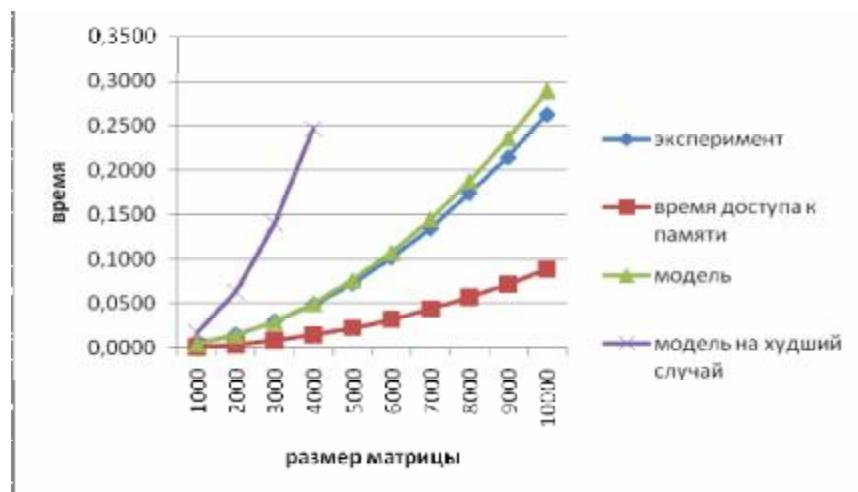
Рис. 6.22. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма умножения матрицы на вектор (блочное разбиение матрицы, 4 потока) с использованием вложенного параллелизма

Как видно из представленных таблиц и графиков, при небольших размерах матрицы и незначительных объемах вычислений, алгоритм, основанный на автоматическом распределении вычислительной нагрузки, существенно проигрывает по производительности алгоритму, основанному на «ручном» разделении данных. Это объясняется тем, что в этом случае больший вес имеют накладные расходы на организацию параллелизма. Однако с ростом вычислительной нагрузки, доля времени, затраченного на организацию параллелизма, становится меньше, производительность алгоритма повышается. При максимальных размерах матрицы и вектора он практически не уступает алгоритму, реализованному на основе «ручного» разделения.

В таблице 6.12 и на рис. 6.23 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матрицы на вектор с использованием четырех потоков со временем  $T_p^*$ , полученным при помощи модели (6.15). Частота кэш промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,067.

**Таблица 6.12.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на блочном разбиении матрицы, с использованием четырех потоков (вложенный параллелизм)

Размер матрицы	$T_p$	$T_p^* (calc)$ (модель)	Модель 6.14 – оценка сверху		Модель 6.15 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	0,0050	0,0020	0,0131	0,0151	0,0009	0,0029
2000	0,0147	0,0078	0,0524	0,0602	0,0035	0,0113
3000	0,0296	0,0174	0,1179	0,1353	0,0079	0,0253
4000	0,0488	0,0308	0,2096	0,2404	0,0140	0,0448
5000	0,0720	0,0479	0,3275	0,3754	0,0219	0,0698
6000	0,1016	0,0688	0,4716	0,5405	0,0316	0,1004
7000	0,1343	0,0935	0,6420	0,7355	0,0430	0,1365
8000	0,1739	0,1220	0,8385	0,9605	0,0562	0,1782
9000	0,2144	0,1542	1,0612	1,2155	0,0711	0,2253
10000	0,2632	0,1903	1,3101	1,5004	0,0878	0,2781



**Рис. 6.23.** График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма от объема исходных данных при использовании четырех потоков (блочное разбиение, вложенный параллелизм)

## 6.8. Краткий обзор главы

В данной Главе на примере задачи умножения матрицы на вектор рассматриваются возможные схемы разделения матриц между потоками параллельной программы, предназначеннной для выполнения на многопроцессорной вычислительной системе с общей памятью и/или на вычислительной системе с многоядерными процессорами. Эти схемы разделения данных являются общими и могут быть использованы для организации параллельных вычислений при выполнении любых матричных операций. В числе излагаемых схем способы разбиения матриц на *полосы* (по вертикали или горизонтали) или на *прямоугольные наборы элементов (блоки)*.

Далее в Главе с использованием рассмотренных способов разделения матриц подробно излагаются три возможных варианта параллельного выполнения операции умножения матрицы на вектор. Первый алгоритм основан на разделении матрицы между потоками по строкам, второй – на разделении матрицы по столбцам, а третий – на блочном разделении данных. Каждый алгоритм представлен в соответствии с общей схемой, описанной в Главе 3, - вначале определяются базовые подзадачи, затем выделяются информационные зависимости подзадач, далее обсуждается масштабирование и распределение подзадач между вычислительными элементами. В завершение для каждого алгоритма проводится анализ эффективности получаемых параллельных вычислений и приводятся результаты вычислительных экспериментов. Для всех рассматриваемых параллельных алгоритмов умножения матрицы на вектор приводятся возможные варианты программной реализации.

Полученные показатели ускорения и эффективности показывают, что все используемые способы разделения данных приводят к равномерной балансировке вычислительной нагрузки, и отличия возникают только в трудоемкости выполняемых информационных взаимодействий между потоками. В этом отношении интересным представляется проследить, как выбор способа разделения данных влияет на характер организации параллельных участков программы, выделить основные различия в использовании потоками общих ресурсов и необходимых операций по синхронизации доступа к ним.

На рис. 6.22 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов. Как можно заметить, некоторое преимущество по ускорению имеет параллельный алгоритм умножения матрицы на вектор при ленточном разделении по строкам.

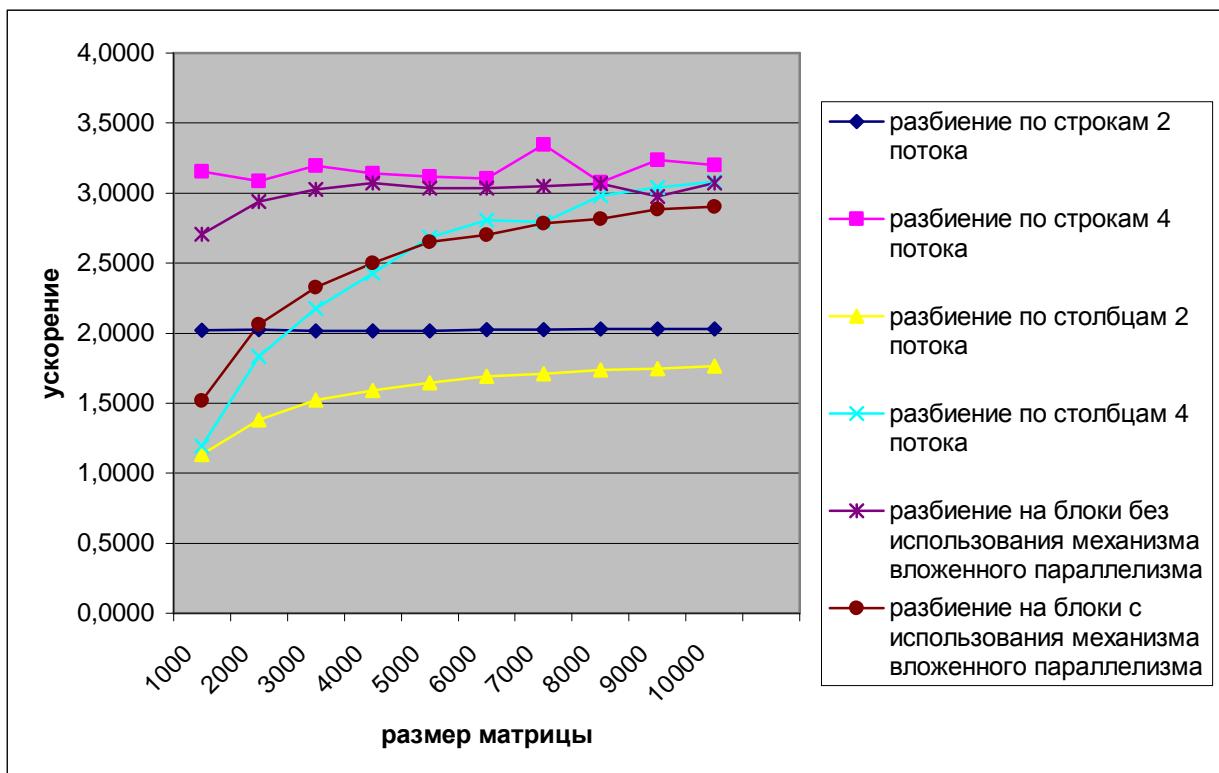


Рис. 6.24. Показатели ускорения рассмотренных параллельных алгоритмов умножения по результатам вычислительных экспериментов

## 6.9. Обзор литературы

Задача умножения матрицы на вектор часто используется как демонстрационный пример параллельного программирования и, как результат, широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы [8,10,72,85]. Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе [50].

## 6.10. Контрольные вопросы

- Назовите основные способы распределения элементов матрицы между потоками.
- В чем состоит постановка задачи умножения матрицы на вектор?
- Какова вычислительная сложность последовательного алгоритма умножения матрицы на вектор?
- Какие подходы могут быть предложены для разработки параллельных алгоритмов умножения матрицы на вектор?
- Представьте общие схемы рассмотренных параллельных алгоритмов умножения матрицы на вектор.
- Проведите анализ и получите показатели эффективности для одного из рассмотренных алгоритмов.
- Какой из представленных алгоритмов умножения матрицы на вектор обладает лучшими показателями ускорения и эффективности?
- Может ли использование циклической схемы разделения данных повлиять на время работы каждого из представленных алгоритмов?
- Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных? В чем различие необходимых операций по организации

параллельных участков программы и синхронизации доступа к общим ресурсам при разделении матрицы по строкам и столбцам?

10. Какие информационные взаимодействия выполняются для блочного алгоритма умножения матрицы на вектор?
11. Какие средства технологии OpenMP и функции соответствующей библиотеки оказались необходимыми при программной реализации алгоритмов?

### **6.11. Задачи и упражнения**

1. Выполните реализацию параллельного алгоритма, основанного на ленточном разбиении матрицы на вертикальные полосы. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими оценками.
2. Выполните реализацию параллельного алгоритма, основанного на разбиении матрицы на блоки. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими оценками.

Глава 7. Параллельные методы матричного умножения.....	1
7.1. Постановка задачи .....	2
7.2. Последовательный алгоритм.....	2
7.2.1. Описание алгоритма.....	2
7.2.2. Анализ эффективности .....	3
7.2.3. Программная реализация .....	4
7.2.4. Результаты вычислительных экспериментов.....	5
7.3. Базовый параллельный алгоритм умножения матриц .....	7
7.3.1. Определение подзадач .....	7
7.3.2. Выделение информационных зависимостей.....	8
7.3.3. Масштабирование и распределение подзадач.....	8
7.3.4. Анализ эффективности .....	8
7.3.5. Программная реализация .....	9
7.3.6. Результаты вычислительных экспериментов .....	9
7.4. Алгоритм умножения матриц, основанный на ленточном разделении данных .....	12
7.4.1. Определение подзадач .....	12
7.4.2. Выделение информационных зависимостей.....	12
7.4.3. Масштабирование и распределение подзадач.....	13
7.4.4. Анализ эффективности .....	13
7.4.5. Программная реализация .....	14
7.4.6. Результаты вычислительных экспериментов .....	15
7.5. Блочный алгоритм умножения матриц .....	17
7.5.1. Определение подзадач .....	17
7.5.2. Выделение информационных зависимостей.....	18
7.5.3. Масштабирование и распределение подзадач.....	18
7.5.4. Анализ эффективности .....	19
7.5.5. Программная реализация .....	19
7.5.6. Результаты вычислительных экспериментов .....	20
7.6. Блочный алгоритм, эффективно использующий кэш-память.....	22
7.6.1. Последовательный алгоритм .....	22
7.6.2. Параллельный алгоритм .....	24
7.6.3. Результаты вычислительных экспериментов .....	25
7.7. Краткий обзор главы .....	27
7.8. Обзор литературы .....	28
7.9. Контрольные вопросы .....	28
7.10. Задачи и упражнения.....	29

## **Глава 7.** **Параллельные методы матричного умножения**

Операция умножения матриц является одной из основных задач матричных вычислений. В данном разделе рассматриваются несколько разных параллельных алгоритмов для выполнения этой операции. Два из них основаны на ленточной схеме разделения данных. Другие два метода используют блочную схему разделения данных, при этом последний из них основывается на разбиении матриц на блоки такого размера, чтобы блоки можно было полностью поместить в кэш-память.

## 7.1. Постановка задачи

Умножение матрицы  $A$  размера  $m \times n$  и матрицы  $B$  размера  $n \times l$  приводит к получению матрицы  $C$  размера  $m \times l$ , каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \leq i < m, 0 \leq j < l. \quad (7.1)$$

Как следует из (7.1), каждый элемент результирующей матрицы  $C$  есть скалярное произведение соответствующих строки матрицы  $A$  и столбца матрицы  $B$ :

$$c_{ij} = (a_i, b_j^T), a_i = (a_{i0}, a_{i1}, \dots, a_{i, m-1}), b_j^T = (b_{0j}, b_{1j}, \dots, b_{n-1, j}). \quad (7.2)$$

Этот алгоритм предполагает выполнение  $m \cdot n \cdot l$  операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера  $n \times n$  количество выполненных операций имеет порядок  $O(n^3)$ . Известны последовательные алгоритмы умножения матриц, обладающие меньшей вычислительной сложностью (например, алгоритм Страссена (*Strassen's algorithm*) [17]), но эти алгоритмы требуют определенных усилий для их освоения и, как результат, в данном разделе при разработке параллельных методов в качестве основы будет использоваться приведенный выше последовательный алгоритм. Также будем предполагать далее, что все матрицы являются квадратными и имеют размер  $n \times n$ .

## 7.2. Последовательный алгоритм

### 7.2.1. Описание алгоритма

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

```
// Алгоритм 7.1
// Последовательный алгоритм умножения матриц
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i, j, k;
...
for (i=0; i<Size; i++){
    for (j=0; j<Size; j++){
        MatrixC[i][j] = 0;
        for (k=0; k<Size; k++){
            MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
        }
    }
}
```

Алгоритм 7.1. Последовательный алгоритм умножения двух квадратных матриц

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы  $C$ . Действительно, при выполнении одной итерации внешнего цикла (цикла по переменной  $i$ ) вычисляется одна строка результирующей матрицы (см. рис. 7.1)

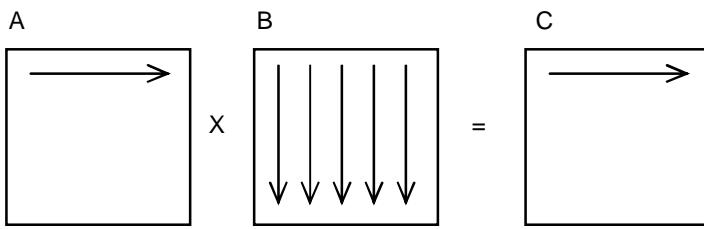


Рис. 7.1. На первой итерации цикла по переменной  $i$  используется первая строка матрицы  $A$  и все столбцы матрицы  $B$  для того, чтобы вычислить элементы первой строки результирующей матрицы  $C$

Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы  $C$  размером  $n \times n$  необходимо выполнить  $n^2 \cdot (2n - 1)$  скалярных операций и затратить время

$$T_1 = n^2 \cdot (2n - 1) \cdot t \quad (7.3)$$

где  $t$  есть время выполнения одной элементарной скалярной операции.

### 7.2.2. Анализ эффективности

При анализе эффективности последовательного алгоритма умножения матриц будем опираться на положения, изложенные в разделе 6.5.4.

Итак, время выполнения алгоритма складывается из времени, которое тратится непосредственно на вычисления, и времени, необходимого на чтение данных из оперативной памяти в кэш процессора. Время вычислений может быть оценено с использованием формулы (7.3).

Теперь необходимо оценить объем данных, которые необходимо прочитать из оперативной памяти в кэш вычислительного элемента в случае, когда размер матриц настолько велик, что они одновременно не могут быть помещены в кэш. Для вычисления одного элемента результирующей матрицы необходимо прочитать в кэш элементы одной строки матрицы  $A$  и одного столбца матрицы  $B$ . Для записи полученного результата дополнительно требуется чтение соответствующего элемента матрицы  $C$  из оперативной памяти. Важно отметить, что приведенные оценки количества читаемых из памяти данных справедливы, если все эти данные отсутствуют в кэше. В реальности часть этих данных может присутствовать в кэше и тогда объем переписываемых данных в кэш уменьшается. Расположение данных в каждом конкретном случае зависит от многих величин (размер кэша, объема обрабатываемых данных, стратегии замещения строк кэша и т.п.). Детальный анализ всех этих моментов является достаточно затруднительным. Возможный выход в таком случае состоит в оценке максимально возможного объема данных, перемещаемых из памяти в кэш (построение оценки сверху). В нашем случае всего необходимо вычислить  $n^2$  элементов результирующей матрицы – тогда, предполагая, что при вычислении каждого очередного элемента требуется прочитать в кэш все необходимые данные, следует, что общий объем данных, необходимых для чтения из оперативной памяти в кэш, не превышает величины  $2n^3 + n^2$ .

Таким образом, оценка времени выполнения последовательного алгоритма умножения матриц может быть представлена следующим образом:

$$T_1 = n^2(2n - 1) \cdot t + 64 \cdot (2n^3 + n^2) / b \quad (7.4)$$

где  $\beta$  есть пропускная способность канала доступа к оперативной памяти (константа 64 введена для учета факта, что в случае кэш-промаха из ОП читается кэш-строка размером 64 байт).

Если, как и в предыдущем разделе, помимо пропускной способности учесть латентность памяти, модель приобретет следующий вид:

$$T_1 = n^2(2n-1) \cdot t + (2n^3 + n^2)(a + 64/b) \quad (7.5)$$

где  $\alpha$  есть латентность оперативной памяти.

Обе предыдущие модели являются моделями на худший случай и дают сильно завышенную оценку времени выполнения алгоритма, так как доступ к оперативной памяти происходит не при каждом обращении к элементу (см. раздел 6.5.4).

Как и ранее, введем в разработанную модель (7.5) величину  $g$ ,  $0 \leq g \leq 1$ , для задания частоты возникновения кэш промахов. Тогда оценка времени выполнения алгоритма матричного умножения принимает вид:

$$T_1 = n^2(2n-1) \cdot t + g(2n^3 + n^2)(a + 64/b) \quad (7.6)$$

### 7.2.3. Программная реализация

Представим возможный вариант последовательной программы умножения матриц.

**1. Главная функция программы.** Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 7.1
// Serial matrix multiplication
void main(int argc, char* argv[]) {
    double* pAMatrix; // The first argument of matrix multiplication
    double* pBMatrix; // The second argument of matrix multiplication
    double* pCMatrix; // The result of matrix multiplication
    int Size;          // Sizes of matricies

    // Data initialization
    ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, Size);

    // Matrix multiplication
    SerialResultCalculation(pAMatrix, pBMatrix, pCMatrix, Size);

    // Program termination
    ProcessTermination(pAMatrix, pBMatrix, pCMatrix, Size);
}
```

**2. Функция ProcessInitialization.** Эта функция определяет размер матриц и элементы для матриц  $A$  и  $B$ , результирующая матрица  $C$  заполняется нулями. Значения элементов для матриц  $A$  и  $B$  определяются в функции RandomDataInitialization.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
                           double* &pCMatrix, int &Size) {
    int i, j; // Loop variables

    do {
        printf ("\nEnter size of the initial matricies: ");
        scanf ("%d", &Size);
        if (Size <= 0) {
            printf ("Size of the matricies must be greater than 0! \n ");
        }
    } while (Size <= 0);

    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++)
            pCMatrix[i*Size+j] = 0;
    RandomDataInitialization(pAMatrix, pBMatrix, Size);
}
```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

### 3. Функция *SerialResultCalculation*. Данная функция производит умножение матриц.

```
// Function for calculating matrix multiplication
void SerialResultCalculation(double* pAMatrix, double* pBMatrix,
    double* pCMatrix, int Size) {
    int i, j, k; // Loop variables
    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++)
            for (k=0; k<Size; k++)
                pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
}
```

Следует отметить, что приведенный программный код может быть оптимизирован (вычисление индексов, использование кэша и т.п.), однако такая оптимизация не является целью данного учебного материала и усложняет понимание программ.

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

#### 7.2.4. Результаты вычислительных экспериментов

Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Известно, что в современных компиляторах реализованы достаточно сложные алгоритмы оптимизации кода: в некоторых случаях может автоматически выполняться развертка циклов, осуществление предсказаний потребности данных и т.п. Для того, чтобы не учитывать влияние этих средств и рассматривать код «как он есть», функция оптимизации кода компилятором была отключена.

Для того, чтобы оценить влияние оптимизации, производимой компилятором, на эффективность приложения, проведем простой эксперимент. Измерим время выполнения оптимизированной и неоптимизированной версий программы, выполняющей последовательный алгоритм умножения матриц для разных размеров матриц. Результаты проведенных экспериментов представлены в таблице 7.1 и на рис. 7.2.

**Таблица 7.1.** Сравнение времени выполнения оптимизированной и неоптимизированной версии последовательного алгоритма умножения матриц

Размер матриц	Компиляторная оптимизация включена	Компиляторная оптимизация выключена
1000,0000	8,2219	24,8192
1500,0000	28,6027	85,8869
2000,0000	75,1572	176,5772
2500,0000	145,2053	403,2405
3000,0000	267,0592	707,1501

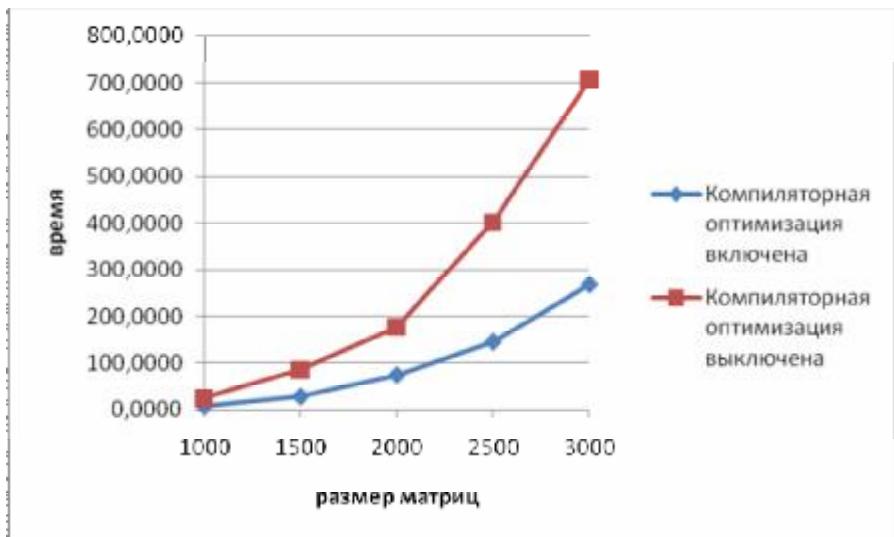


Рис. 7.2. Графики зависимости времени выполнения оптимизированной и неоптимизированной версий последовательного алгоритма

Как видно из представленных графиков, оптимизация кода при помощи компилятора позволяет добиться более чем двукратного ускорения без каких-либо усилий со стороны программиста.

Для того, чтобы оценить время одной операции  $\tau$ , измерим время выполнения последовательного алгоритма умножения матриц при малых объемах данных, таких, чтобы три матрицы, участвующие в умножении, полностью поместились в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицы-аргументы случайными числами, а матрицу-результат – нулями. Выполнение этих действий гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение  $\tau$ , равное 6,402 нс.

Оценка времени латентности  $\alpha$  и величины пропускной способности канала доступа к оперативной памяти  $\beta$  проводилась в п. 6.5.4 и определена для используемого вычислительного узла как  $\beta = 12,44$  Гб/с и  $\alpha = 8,31$  нс.

В таблице 7.2 и на рис. 7.3 представлены результаты сравнения времени выполнения последовательного алгоритма умножения матриц со временем, полученным при помощи модели (7.6). Как следует из приведенных данных, погрешность аналитической оценки трудоемкости алгоритма матричного умножения уменьшается при увеличении размера матриц (для  $n=3000$  относительная погрешность составляет менее 1%). Частота кэш промахов, измеренная с помощью системы VPS оказалась равной 0,505.

**Таблица 7.2.** Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма умножения матриц

Размер матриц	Эксперимент	Время счета	Время доступа к памяти	Модель
1000	24,8192	12,7976	13,2390	26,0366
1500	85,8869	43,1991	44,6742	87,8733
2000	176,5772	102,4064	105,8855	208,2919
2500	403,2405	200,0225	206,7973	406,8198
3000	707,1501	345,6504	357,3339	702,9843

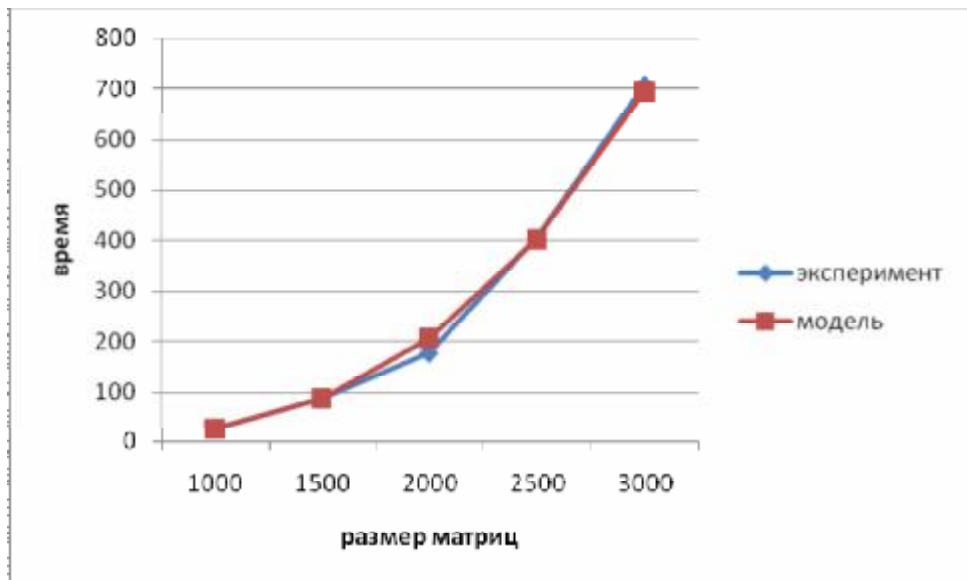


Рис. 7.3. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных

### 7.3. Базовый параллельный алгоритм умножения матриц

Рассмотрим параллельный алгоритм умножения матриц, в основу которого будет положено разбиение матрицы  $A$  на непрерывные последовательности строк (*горизонтальные полосы*).

#### 7.3.1. Определение подзадач

Из определения операции матричного умножения следует, что вычисление всех элементов матрицы  $C$  может быть выполнено независимо друг от друга. Как результат, возможный подход для организации параллельных вычислений состоит в использовании в качестве базовой подзадачи процедуры определения одного элемента результирующей матрицы  $C$ . Для проведения всех необходимых вычислений каждая подзадача должна производить вычисления над элементами одной строки матрицы  $A$  и одного столбца матрицы  $B$ . Общее количество получаемых при таком подходе подзадач оказывается равным  $n^2$  (по числу элементов матрицы  $C$ ).

Рассмотрев предложенный подход, можно отметить, что достигнутый уровень параллелизма является в некоторой степени избыточным. Обычно при проведении практических расчетов количество сформированных подзадач превышает число имеющихся вычислительных элементов (процессоров и/или ядер) и, как результат, неизбежным является этап укрупнения базовых задач. В этом плане может оказаться полезным агрегация вычислений уже на шаге выделения базовых подзадач. Возможное решение может состоять в объединении в рамках одной подзадачи всех вычислений, связанных не с одним, а с несколькими элементами результирующей матрицы  $C$ . Для дальнейшего рассмотрения в рамках данного подраздела определим базовую задачу как процедуру вычисления всех элементов одной из строк матрицы  $C$ . Такой подход приводит к снижению общего количества подзадач до величины  $n$ .

Для выполнения всех необходимых вычислений базовой подзадаче должны быть доступны одна из строк матрицы  $A$  и все столбцы матрицы  $B$ . Простое решение этой проблемы – дублирование матрицы  $B$  во всех подзадачах. Следует отметить, что такой подход не приводит к реальному дублированию данных, поскольку разрабатываемый алгоритм ориентирован на применение для вычислительных систем с общей разделяемой памятью, к которой имеется доступ со всех используемых вычислительных элементов.

### 7.3.2. Выделение информационных зависимостей

Для вычисления одной строки матрицы  $C$  необходимо, чтобы в каждой подзадаче содержалась строка матрицы  $A$  и был обеспечен доступ ко всем столбцам матрицы  $B$ . Способ организации параллельных вычислений представлен на рис. 7.4.

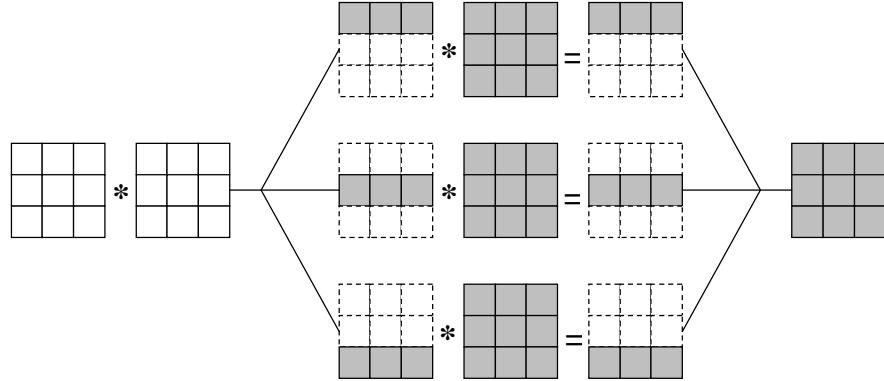


Рис. 7.4. Организация вычислений при выполнении параллельного алгоритма умножения матриц, основанного на разделении матриц по строкам

### 7.3.3. Масштабирование и распределение подзадач

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае, когда размер матриц  $n$  оказывается больше, чем число вычислительных элементов (процессоров и/или ядер)  $p$ , базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних строк матрицы. В этом случае, исходная матрица  $A$  и матрица-результат  $C$  разбиваются на ряд горизонтальных полос. Размер полос при этом следует выбрать равным  $k=n/p$  (в предположении, что  $n$  кратно  $p$ ), что позволит по-прежнему обеспечить равномерность распределения вычислительной нагрузки по вычислительным элементам.

### 7.3.4. Анализ эффективности

Будем проводить анализ эффективности реализации базового параллельного алгоритма умножения матриц по схеме, изложенной в разделе 6.5.4.

Данный параллельный алгоритм обладает хорошей «локальностью вычислений». Это означает, что данные, которые обрабатывает один из потоков параллельной программы, не изменяются другим потоком. Нет взаимодействия между потоками, нет необходимости в синхронизации. Значит, для того, чтобы определить время выполнения параллельного алгоритма, необходимо знать, сколько вычислительных операций выполняет каждый поток параллельной программы (вычисления выполняются потоками параллельно) и сколько данных необходимо прочитать из оперативной памяти в кэш процессора (доступ к памяти осуществляется строго последовательно).

Для вычисления одного элемента результирующей матрицы необходимо выполнить скалярное умножение строки матрицы  $A$  на столбец матрицы  $B$ . Выполнение скалярного умножения включает  $(2n-1)$  вычислительных операций. Каждый поток вычисляет элементы горизонтальной полосы результирующей матрицы, число элементов в полосе составляет  $n^2/p$ . Таким образом, время, которое тратится на вычисления, может быть определено по формуле:

$$T_{calc} = (n^2 / p)(2n-1) \cdot t . \quad (7.7)$$

Для оценки объема данных, которые необходимо прочитать из оперативной памяти в кэш, снова применим подход, изложенный в п. 7.2.3. Для вычисления одного элемента результирующей матрицы  $C$  необходимо прочитать в кэш  $n+8n+8$  элементов данных. Каждый поток вычисляет  $n/p$  элементов матрицы  $C$ , однако для определения полного

объема переписываемых в кэш данных следует учитывать, что чтение значений из оперативной памяти может выполняться только последовательно (см. п. 6.5.4 и рис. 6.6). Как результат, сокращение объема переписываемых в кэш данных достигается только для матрицы  $B$  (прочитанный однократно в кэш столбец матрицы  $B$  может использоваться всеми потоками без повторного чтения из оперативной памяти). Чтение же строк матрицы  $A$  и элементов матрицы  $C$  в предельном случае должно быть выполнено полностью и последовательно. Как результат, время работы с оперативной памятью при выполнении описанного параллельного алгоритма умножения матриц может быть определено в соответствии со следующим соотношением:

$$T_{mem} = \left( n^3 + n^3/p + n^2 \right) (a + 64/b), \quad (7.8)$$

где, как и ранее,  $\beta$  есть пропускная способность канала доступа к оперативной памяти, а  $\alpha$  латентность оперативной памяти.

Следовательно, время выполнения параллельного алгоритма составляет:

$$T_p = (n^2/p)(2n-1) \cdot t + \left( n^3 + n^3/p + n^2 \right) (a + 64/b) \quad (7.9)$$

Как и ранее, следует учесть, что часть необходимых данных может быть перемещена в кэш заблаговременно при помощи тех или иных механизмов предсказания. Кроме того, обращение к данным не обязательно приводит к кэш промаху и, соответственно, к чтению данных из оперативной памяти (необходимые данные могут находиться и в кэш памяти). Данные факторы можно, как и в предыдущих случаях, учесть при помощи введения в модель показатель частоты кэш промахов:

$$T_p = (n^2/p)(2n-1) \cdot t + g \left( n^3 + n^3/p + n^2 \right) (a + 64/b). \quad (7.10)$$

### 7.3.5. Программная реализация

Для того, чтобы разработать параллельную программу, реализующую описанный подход, при помощи технологии OpenMP, необходимо внести минимальные изменения в функцию умножения матриц. Достаточно добавить одну директиву *parallel for* в функции *SerialResultCaclualtion* (назовем новый вариант функции *ParallelResultCaclualtion*):

```
// Программа 7.2
// Function for parallel calculating matrix multiplication
void ParallelResultCalculation(double* pAMatrix, double* pBMatrix,
    double* pCMatrix, int Size) {
    int i, j, k; // Loop variables
#pragma omp parallel for private (j, k)
    for (i=0; i<Size; i++)
        for (j=0; j<Size; j++)
            for (k=0; k<Size; k++)
                pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
}
```

Данная функция производит умножение строк матрицы  $A$  на столбцы матрицы  $B$  с использованием нескольких параллельных потоков. Каждый поток выполняет вычисления над несколькими соседними строками матрицы  $A$  и, таким образом, получает несколько соседних строк результирующей матрицы  $C$ .

### 7.3.6. Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности параллельного алгоритма матричного умножения. Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

Результаты вычислительных экспериментов приведены в таблице 7.3. Времена выполнения алгоритмов указаны в секундах.

**Таблица 7.3.** Результаты вычислительных экспериментов для параллельного алгоритма умножения матриц при ленточной схеме разделении данных по строкам

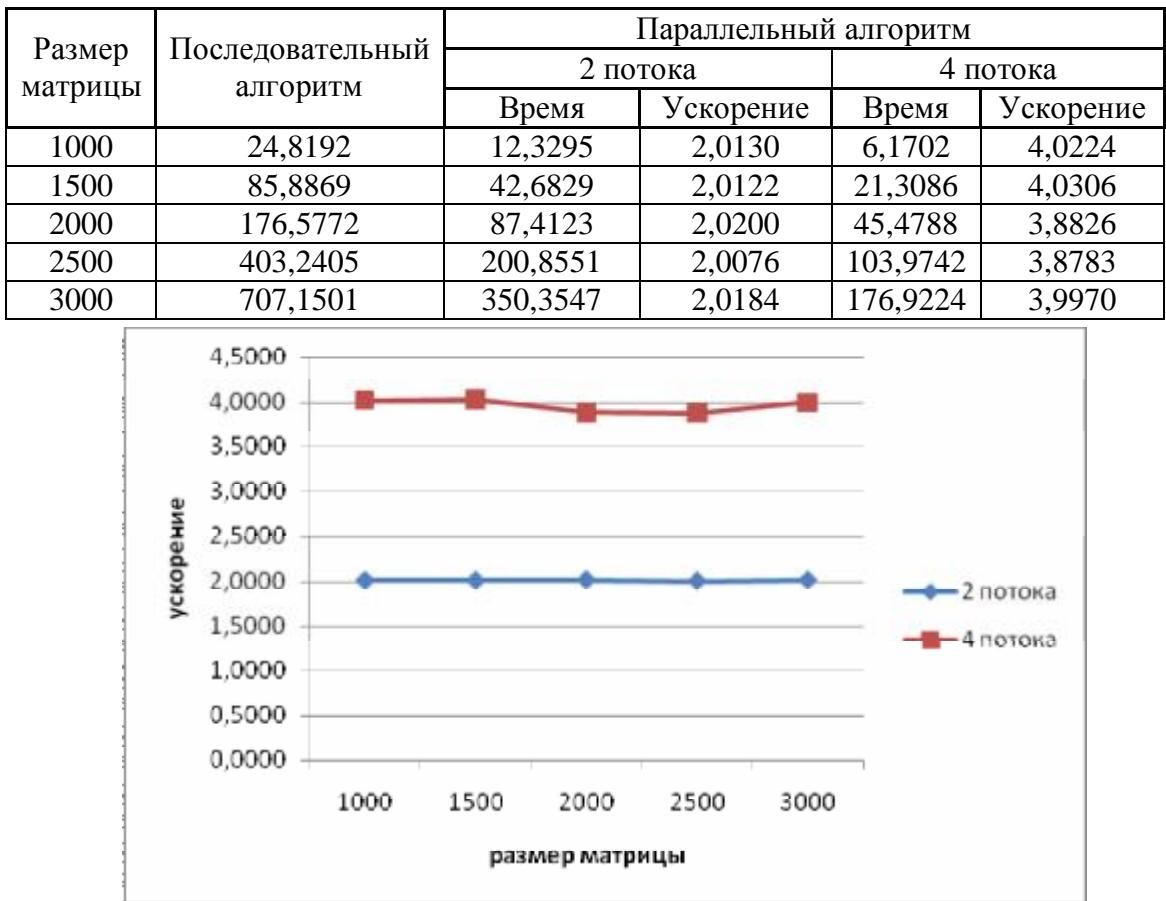


Рис. 7.5. Зависимость ускорения от количества исходных данных при выполнении базового параллельного алгоритма умножения матриц

Можно отметить, что выполненные эксперименты показывают почти идеальное ускорение вычислений для разработанного параллельного алгоритма умножения матриц (и данный результат достигнут в результате незначительной корректировки исходно последовательной программы).

В таблицах 7.4 и 7.5 и на рис. 7.6 и 7.7 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матриц с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (7.4). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,3371, а для четырех потоков значение этой величины была оценена как 0,1832.

**Таблица 7.4.** Сравнение экспериментального и теоретического времени выполнения базового параллельного алгоритма умножения матриц с использованием двух потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 7.4 – оценка сверху		Модель 7.5 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	12,3295	6,3988	19,6652	26,0640	6,6291	13,0279
1500	42,6829	21,5995	66,3552	87,9547	22,3683	43,9679
2000	87,4123	51,2032	157,2688	208,4720	53,0153	104,2185

2500	200,8551	100,0112	307,1452	407,1565	103,5387	203,5499
3000	350,3547	172,8252	530,7234	703,5486	178,9069	351,7321

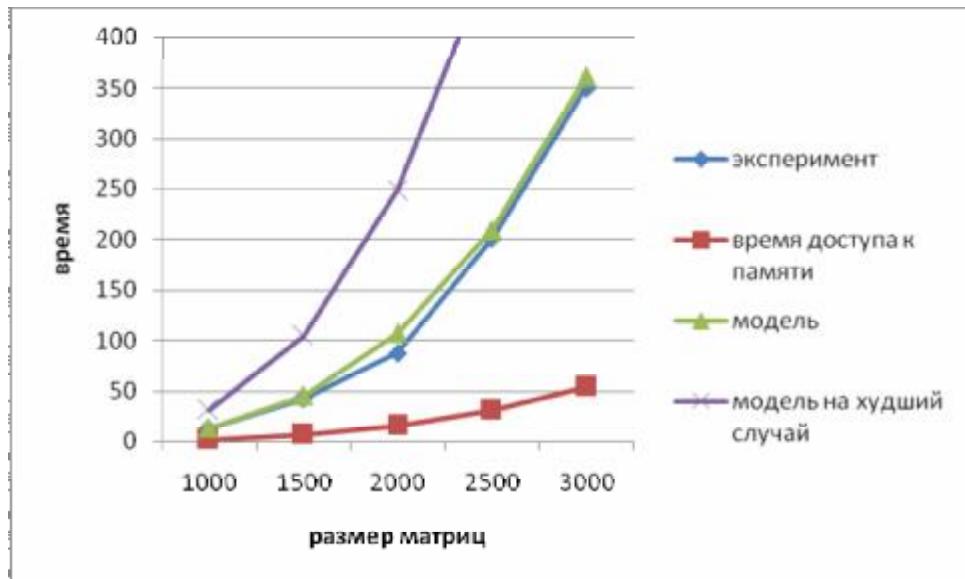


Рис. 7.6. График зависимости экспериментального и теоретического времени выполнения базового параллельного алгоритма от объема исходных данных при использовании двух потоков

**Таблица 7.5.** Сравнение экспериментального и теоретического времени выполнения базового параллельного алгоритма умножения матриц с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 7.4 – оценка сверху		Модель 7.5 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
1000	6,1702	3,1994	16,3898	19,5892	3,0026	6,2020
1500	21,3086	10,7998	55,3009	66,1007	10,1311	20,9309
2000	45,4788	25,6016	131,0661	156,6677	24,0113	49,6129
2500	103,9742	50,0056	255,9680	305,9736	46,8933	96,8990
3000	176,9224	86,4126	442,2892	528,7018	81,0274	167,4400

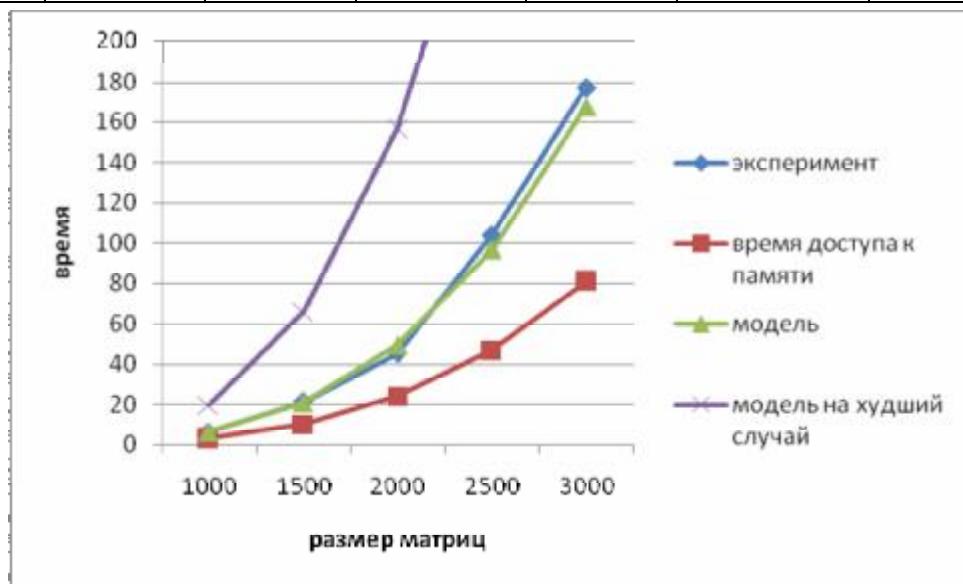


Рис. 7.7. График зависимости экспериментального и теоретического времени выполнения базового параллельного алгоритма от объема исходных данных при использовании четырех потоков

## 7.4. Алгоритм умножения матриц, основанный на ленточном разделении данных

В рассмотренном в подразделе 7.3 только одна из перемножаемых матриц – матрица  $A$  – распределялась между параллельно выполняемыми потоками. В данном подразделе излагается алгоритм, в котором ленточная схема разделения данных применяется и для второй перемножаемой матрицы  $B$  – такой подход, в частности, позволяет улучшить локализацию данных в потоках и повысить эффективность использования кэша.

### 7.4.1. Определение подзадач

Как и ранее, в качестве базовой подзадачи будем рассматривать процедуру определения одного элемента результирующей матрицы  $C$ . Общее количество получаемых при таком подходе подзадач оказывается равным  $n^2$  (по числу элементов матрицы  $C$ ). Выше было отмечено, что достигнутый таким образом уровень параллелизма зачастую является избыточным – количество базовых подзадач существенно превышает число доступных вычислительных элементов. В этом случае необходимо выполнить укрупнение подзадач. В рамках данного подраздела определим базовую подзадачу как процедуру вычисления всех элементов прямоугольного блока матрицы  $C$  (блочная схема разбиения матриц подробно рассмотрена в подразделе 6.2).

При дальнейшем изложении материала для снижения сложности и упрощения получаемых соотношений будем полагать, что число блоков в матрице  $C$  по горизонтали и по вертикали совпадает. Для эффективного выполнения параллельного алгоритма умножения матриц целесообразно выделить число параллельных потоков совпадающим с количеством блоков матрицы  $C$ , т.е. такое количество потоков, которое является полным квадратом  $q^2$  ( $p = q^2$ ). Дополнительно можно отметить заранее, что для эффективного выполнения вычислений количество потоков  $p$  должно быть, по крайней мере, кратным числу вычислительных элементов (процессоров и/или ядер)  $r$ .

### 7.4.2. Выделение информационных зависимостей

Для вычисления одного элемента  $c_{ij}$  результирующей матрицы необходимо выполнить скалярное умножение  $i$ -ой строки матрицы  $A$  и  $j$ -ого столбца матрицы  $B$ . Следовательно, для вычисления всех элементов прямоугольного блока результирующей матрицы

$$C_{i_1-i_2, j_1-j_2} = \{c_{ij} : i_1 \leq i \leq i_2, j_1 \leq j \leq j_2\}$$

необходимо выполнить скалярное умножение строк матрицы  $A$  с индексами  $i$  ( $i_1 \leq i \leq i_2$ ) на столбцы матрицы  $B$  с индексами  $j$  ( $j_1 \leq j \leq j_2$ ). То есть необходимо разделить между потоками параллельной программы как строки матрицы  $A$ , так и столбцы матрицы  $B$ . Для этого воспользуемся механизмом вложенного параллелизма, который был подробно рассмотрен в разделе 6.7.2.

Пусть каждое новое объявление параллельной секции разделяет поток выполнения на  $q$  потоков. В этом случае разделение итераций внешнего цикла матричного умножения между потоками параллельной программы разделит матрицу  $A$  на  $q$  горизонтальных полос. При последующем разделении итераций внутреннего цикла с помощью механизма вложенного параллелизма матрица  $B$  окажется разделенной на  $q$  вертикальных полос (рис. 7.8).

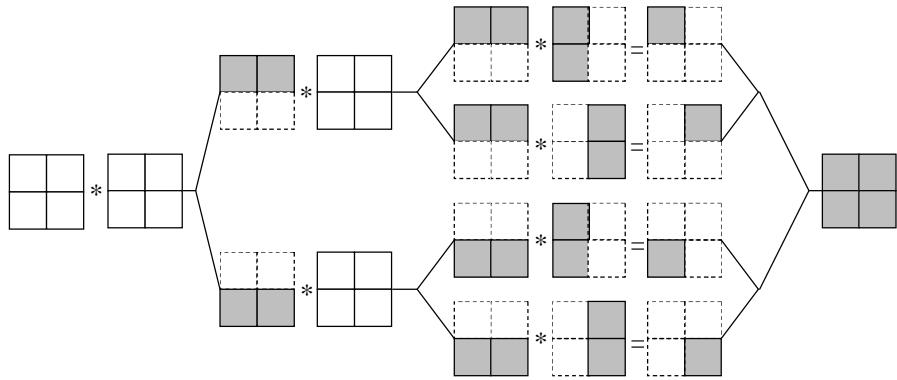


Рис. 7.8. Организация параллельных вычислений при выполнении параллельного алгоритма умножения матриц, основанного на ленточном разделении данных, и использованием четырех потоков

После выполнения вычислений над определенными полосами каждый поток параллельной программы вычислит все элементы блока результирующей матрицы.

#### 7.4.3. Масштабирование и распределение подзадач

Размер блоков матрицы  $C$  может быть подобран таким образом, чтобы общее количество базовых подзадач совпадало с числом выделенных потоков  $\pi$ . Так, например, если определить размер блочной решетки матрицы  $C$  как  $\pi = q \times q$ , то

$$k = m/q, \quad l = n/q,$$

где  $k$  и  $l$  есть количество строк и столбцов в блоках матрицы  $C$ . Такой способ определения размера блоков приводит к тому, что объем вычислений в каждой подзадаче является равным и, тем самым, достигается полная балансировка вычислительной нагрузки между вычислительными элементами.

#### 7.4.4. Анализ эффективности

Пусть для параллельного выполнения операции матричного умножения используется  $\pi$  параллельных потоков ( $\pi = q \times q$ ). Каждый поток вычисляет элементы прямоугольного блока результирующей матрицы, для вычисления каждого элемента необходимо выполнить скалярное произведение строки матрицы  $A$  на столбец матрицы  $B$ . Следовательно, количество операций, которые выполняет каждый поток, составляет  $n^2 \cdot (2n - 1) / p$ .

Для выполнения программы используется  $p$  вычислительных элементов. Если потоки параллельной программы могут быть равномерно распределены по вычислительным элементам, тогда время выполнения вычислений составляет:

$$T_{calc} = (n^2 / p)(2n - 1) \cdot t .$$

Далее оценим объем данных, которые необходимо прочитать из оперативной памяти в кэш процессора. Каждый поток выполняет умножение горизонтальной полосы матрицы  $A$  на вертикальную полосу матрицы  $B$  для того, чтобы получить прямоугольный блок результирующей матрицы. Как и ранее, для вычисления одного элемента результирующей матрицы  $C$  необходимо прочитать в кэш  $2n + 2$  элементов данных. Каждый поток вычисляет  $n^2 / q^2$  элементов матрицы  $C$ , однако для определения полного объема переписываемых в кэш данных следует учитывать, что чтение значений из оперативной памяти может выполняться только последовательно (см. п. 6.5.4 и рис. 6.6). Элементы матриц  $A$  и  $C$ , обрабатываемые в разных потоках, не пересекаются и, в предельном случае, должны читаться в кэш для каждой итерации алгоритма повторно (т.е.  $n^2$  раз). С другой стороны, если кэш память является общей для потоков, то столбцы матрицы  $B$  могут быть использованы без повторного чтения из оперативной памяти (каждый столбец матрицы  $B$

обрабатывается  $q$  потоками, отвечающими за вычисление одного и того вертикального ряда блоков матрицы  $C$ ). Таким образом, время, необходимое на чтение необходимых данных из оперативной памяти составляет:

$$T_{mem} = \left( n^3 + n^3/q + n^2 \right) (a + 64/b),$$

где  $\beta$  есть пропускная способность канала доступа к оперативной памяти, а  $\alpha$  есть латентность оперативной памяти.

Следует обратить внимание на то, что при выполнении представленного алгоритма, реализованного с помощью вложенного параллелизма, «внутренние» параллельные секции создаются и закрываются  $n/q$  раз. На выполнение функций библиотеки OpenMP, поддерживающих вложенный параллелизм, тратится дополнительное время. Кроме того, поскольку для работы этих функций необходимо читать в кэш служебные данные, «полезные» данные будут вытесняться, а затем повторно загружаться в кэш, что также ведет к росту накладных расходов. Как и ранее, величину накладных расходов на организацию и закрытие одной параллельной секции обозначим через  $d$ .

Таким образом, оценка времени выполнения параллельного алгоритма матричного умножения в худшем случае может быть определена следующим образом:

$$T_p = (n^2/p) \cdot (2n-1)t + \left( n^3 + n^3/q + n^2 \right) (a + 64/b) + (n/q) \cdot d \quad (7.11)$$

Если же учесть частоту кэш промахов  $g$ , то выражение (7.11) принимает вид:

$$T_p = (n^2/p) \cdot (2n-1)t + g \left( n^3 + n^3/q + n^2 \right) (a + 64/b) + (n/q) \cdot d \quad (7.12)$$

#### 7.4.5. Программная реализация

Использование механизма вложенного параллелизма OpenMP позволяет существенно упростить реализацию алгоритма. Однако следует отметить, что на данный момент не все компиляторы, реализующие стандарт OpenMP, поддерживают вложенный параллелизм. Для компиляции представленного ниже кода использовался компилятор Intel C++ Compiler 10.0 for Windows, поддерживающий вложенный параллелизм.

Для того, чтобы разработать параллельную программу, реализующую описанный подход, при помощи технологии OpenMP, прежде всего, необходимо включить поддержку вложенного параллелизма при помощи вызова функции *omp\_set\_nested*.

Для разделения матрицы  $A$  на горизонтальные полосы нужно разделить итерации внешнего цикла при помощи директивы *omp parallel for*. Далее при помощи той же директивы необходимо разделить итерации внутреннего цикла для разделения матрицы  $B$  на вертикальные полосы.

```
// Программа 7.3
// Function for parallel calculating matrix multiplication
void ParallelResultCalculation(double* pAMatrix, double* pBMatrix,
    double* pCMatrix, int Size) {
    int i, j, k; // Loop variables
    int NestedThreadsNum = 2;
    omp_set_nested(true);
    omp_set_num_threads(NestedThreadsNum);
#pragma omp parallel for private (j, k)
    for (i=0; i<Size; i++)
#pragma omp parallel for private (k)
        for (j=0; j<Size; j++)
            for (k=0; k<Size; k++)
                pCMatrix[i*Size+j] += pAMatrix[i*Size+k]*pBMatrix[k*Size+j];
}
```

Данная функция производит умножение строк матрицы  $A$  на столбцы матрицы  $B$  с использованием нескольких параллельных потоков. Каждый поток выполняет вычисления над элементами горизонтальной полосы матрицы  $A$  и элементами вертикальной полосы

матрицы  $B$ , таким образом, получает значения элементов прямоугольного блока результирующей матрицы  $C$ .

Отметим, что в приведенной программе для задания количества потоков, создаваемых на каждом уровне вложенности параллельных областей, используется переменная *NestedThreadsNum* (в данном варианте программы ее значение устанавливается равным 2 – данное значение должно переустанавливаться для определения необходимого числа потоков).

#### 7.4.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при ленточном разделении данных проводились при условиях, указанных в п. 7.3.6. Результаты вычислительных экспериментов приведены в таблице 7.6. Времена выполнения алгоритма указаны в секундах.

**Таблица 7.6.** Результаты вычислительных экспериментов для параллельного алгоритма умножения матриц при ленточной схеме разделения данных

Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	24,82	5,6766	4,37
1500	85,89	20,4516	4,20
2000	176,58	44,3016	3,99
2500	403,24	98,5983	4,09
3000	707,15	173,3591	4,08

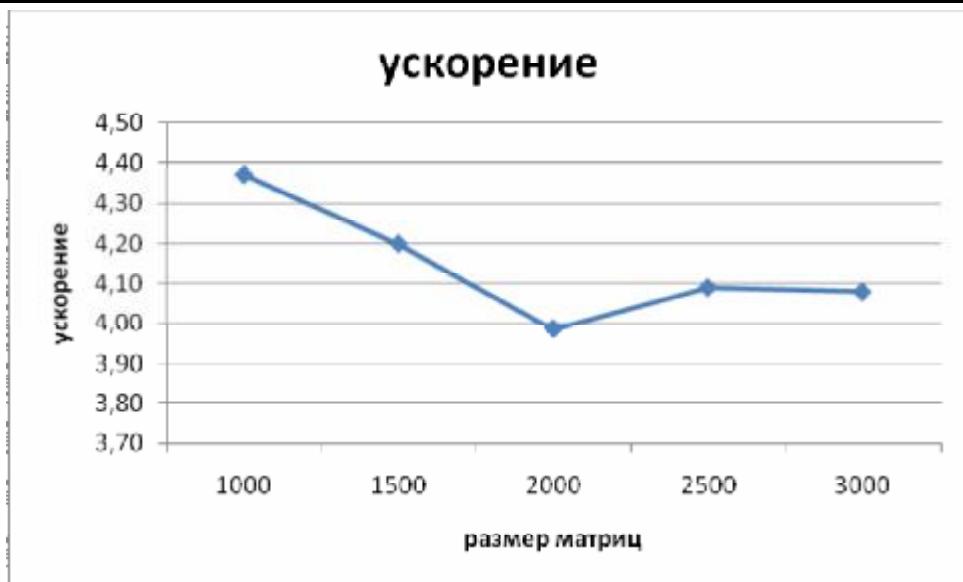


Рис. 7.9. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма умножения матриц, основанного на ленточном разделении матриц

Для того, чтобы оценить величину накладных расходов на организацию и закрытие параллельных секций на каждой итерации внешнего цикла, разработаем еще одну реализацию параллельного алгоритма умножения матриц, также основанного на разделении матриц на полосы. Теперь реализуем это разделение явным образом без использования механизма вложенного параллелизма.

```
// Программа 7.4
// Function for parallel calculating matrix multiplication
void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
    double* pCMatrix, int Size) {
```

```

int BlocksNum = 2;
omp_set_num_threads(BlocksNum*BlocksNum);
#pragma omp parallel
{
    int ThreadID = omp_get_thread_num();
    intRowIndex = ThreadID/BlocksNum;
    int ColIndex = ThreadID%BlocksNum;
    int BlockSize = Size/BlocksNum;
    for (int i=0; i<BlockSize; i++)
        for (int j=0; j<BlockSize; j++)
            for (int k=0; k<Size; k++)
                pCMatrix[(RowIndex*BlockSize+i)*Size + (ColIndex*BlockSize+j)] += 
                    pAMatrix[(RowIndex*BlockSize+i)*Size+k] * 
                    pBMatrix[k*Size+(ColIndex*BlockSize+j)];
}
}

```

Как и ранее в главе 6, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, принято равным 0,25 мкс.

В таблице 7.7 и на рис. 7.10 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матриц с использованием четырех потоков со временем  $T_p^*$ , полученным при помощи модели (7.12). Частота кэш промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,2.

**Таблица 7.7.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матриц, основанного на ленточном разделении данных, с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 7.11 – оценка сверху		Модель 7.12 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
1000	5,6766	3,1995	16,3898	19,5893	3,2780	6,4774
1500	20,4516	10,7999	55,3009	66,1008	11,0602	21,8600
2000	44,3016	25,6017	131,0661	156,6678	26,2132	51,8149
2500	98,5983	50,0058	255,9680	305,9738	51,1936	101,1994
3000	173,3591	86,4128	442,2892	528,7019	88,4578	174,8706

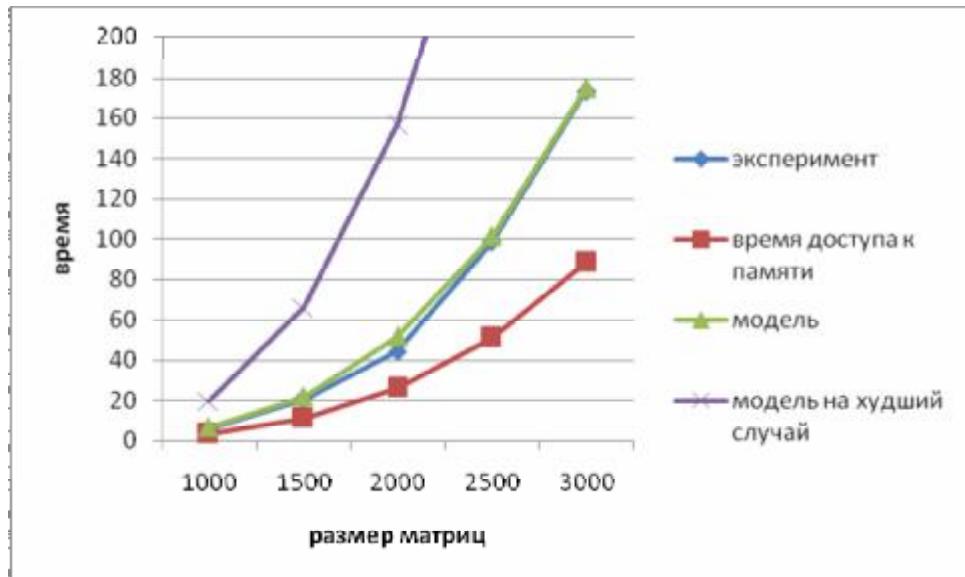


Рис. 7.10. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма, основанного на ленточном разделении данных, от объема исходных данных при использовании четырех потоков

## 7.5. Блочный алгоритм умножения матриц

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Выполним более подробное рассмотрение данного способа организации вычислений. В этом случае не только результирующая матрица, но и матрицы-аргументы матричного умножения разделяются между потоками параллельной программы на прямоугольные блоки. Такой подход позволяет добиться большей локализации данных и повысить эффективность использования кэш памяти.

### 7.5.1. Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 6.2. При таком способе разделения данных исходные матрицы  $A$ ,  $B$  и результирующая матрица  $C$  представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера  $n \times n$ , количество блоков по горизонтали и вертикали являются одинаковым и равным  $q$  (т.е. размер всех блоков равен  $k \times k$ ,  $k = n/q$ ). При таком представлении данных операция матричного умножения матриц  $A$  и  $B$  в блочном виде может быть представлена в виде:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0q-1} \\ \dots & \dots & \dots & \dots \\ A_{q-10} & A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0q-1} \\ \dots & \dots & \dots & \dots \\ B_{q-10} & B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0q-1} \\ \dots & \dots & \dots & \dots \\ C_{q-10} & C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix}, \quad (7.13)$$

где каждый блок  $C_{ij}$  матрицы  $C$  определяется в соответствии с выражением

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj}. \quad (7.14)$$

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы  $C$ .

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы  $A$  и столбцов матрицы  $B$ .

Широко известны параллельные алгоритмы умножения матриц, основанные на блочном разделении данных, ориентированные на многопроцессорные вычислительные системы с распределенной памятью [10]. При разработке алгоритмов, ориентированных на использование параллельных вычислительных систем с распределенной памятью следует учитывать, что размещение всех требуемых данных в каждой подзадаче (в данном случае – размещение в подзадачах необходимых наборов строк матрицы  $A$  и столбцов матрицы  $B$ ) неизбежно приведет к дублированию и к значительному росту объема используемой памяти. Как результат, вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы при помощи передачи сообщений. К числу алгоритмов, реализующих описанный подход, относятся *алгоритм Фокса* (*Fox*) и *алгоритм Кэннона* (*Cannon*). Отличие этих алгоритмов состоит в последовательности передачи матричных блоков между процессорами вычислительной системы.

При выполнении параллельных алгоритмов на системах с общей памятью передача данных между процессорами уже не требуется. Различия между параллельными алгоритмами в этом случае состоят в порядке организации вычислений над матричными блоками, удовлетворяющие соотношению (7.11). Возможная естественным образом определяемая вычислительная схема рассматривается в п. 7.5.2.

### 7.5.2. Выделение информационных зависимостей

За основу параллельных вычислений для матричного умножения при блочном разделении данных принимается подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы  $C$  и при этом подзадачи на каждой итерации расчетов обрабатывают только по одному блоку исходных матриц  $A$  и  $B$ . Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы  $C$ , т.е. подзадача  $(i,j)$  отвечает за вычисление блока  $C_{ij}$  – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы  $C$ .

Как уже отмечалось выше, для вычисления блока результирующей матрицы поток должен выполнить умножение горизонтальной полосы матрицы  $A$  на вертикальную полосу матрицы  $B$ . Организуем поблочное умножение полос. На каждой итерации  $i$  алгоритма  $i$ -ый блок полосы матрицы  $A$  умножается на  $i$ -ый блок полосы матрицы  $B$ , результат умножения блоков прибавляется к блоку результирующей матрицы (рис. 7.11). Количество итераций определяется размером блочной решетки.

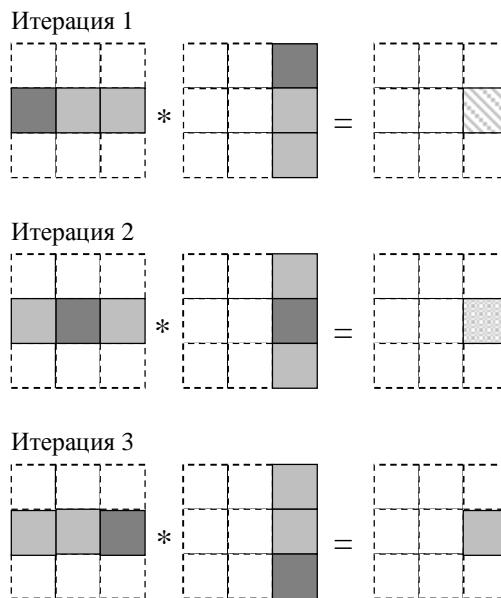


Рис. 7.11. Схема организации блочного умножения полос

### 7.5.3. Масштабирование и распределение подзадач

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом вычислительных элементов  $p$ . Так, например, в наиболее простом случае, когда число вычислительных элементов представимо в виде  $p=s^2$  (т.е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали равным  $s$  (т.е.  $q=s$ ). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и, тем самым, достигается полная балансировка вычислительной нагрузки между вычислительными элементами. В случае, когда число вычислительных элементов не является полным квадратом, число базовых подзадач  $\pi=q \cdot q$  должно быть, по крайней мере, кратно числу вычислительных элементов.

#### 7.5.4. Анализ эффективности

При анализе эффективности параллельного алгоритма умножения матриц, основанного на разделении данных на блоки, можно использовать оценки, полученные при анализе предыдущего алгоритма. Действительно, каждый поток выполняет умножение полос матриц-аргументов. Блоchное разбиение полос вносит изменение лишь в порядок выполнения вычислений, общий же объем вычислительных операций при этом не изменяется. Поэтому вычислительная сложность блочного алгоритма составляет:

$$T_{calc} = (n^2 / p)(2n - 1) \cdot t.$$

Оценим объем данных, которые необходимо прочитать из оперативной памяти в кэш процессора. На каждой итерации алгоритма каждый поток выполняет умножение двух матричных блоков размером  $(n/q) \times (n/q)$ . Для оценки объема данных, который при выполнении этой операции должен быть прочитан из оперативной памяти в кэш, можно воспользоваться формулой (7.5), полученной при анализе эффективности последовательного алгоритма, с поправкой на размер матричного блока.

$$T_{mem}^1 = (2 \cdot (n/q)^3 + (n/q)^2)(a + 64/b)$$

Поскольку для вычисления блока результирующей матрицы каждый поток выполняет  $q$  итераций, то для определения времени, которое каждый поток тратит на чтение необходимых данных в кэш, необходимо умножить величину  $T_{mem}^1$  на число итераций  $q$ . Обращение нескольких потоков к памяти происходит строго последовательно. Общее число потоков –  $q^2$ . Таким образом, время считывания данных из оперативной памяти в кэш для блочного алгоритма умножения матриц составляет:

$$T_{mem} = (2 \cdot (n/q)^3 + (n/q)^2)(a + 64/b) \cdot q \cdot q^2.$$

Однако при более подробном анализе алгоритма можно заметить, что блок, который считывается в кэш одним из потоков, одновременно используется и другими ( $q-1$ ) потоками. Так, например, при использовании решетки потоков  $2 \times 2$ , при выполнении первой итерации алгоритма блок  $A_{11}$  используется одновременно нулевым и первым потоком, блок  $A_{21}$  – одновременно вторым и третьим потоком, блок  $B_{11}$  – нулевым и вторым, а блок  $B_{12}$  – первым и третьим потоками. Следовательно, время, необходимое на чтение данных из оперативной памяти в кэш составляет:

$$T_{mem} = (2 \cdot (n/q)^3 + (n/q)^2)(a + 64/b) \cdot q \cdot q = (2 \cdot n^3/q + n^2)(a + 64/b).$$

Как итог, общее время выполнения параллельного алгоритма умножения матриц, основанного на блочном разделении данных в худшем случае, определяется соотношением:

$$T_p = (n^2 / p)(2n - 1) \cdot t + (2 \cdot n^3/q + n^2)(a + 64/b) \quad (7.15)$$

Если же учесть частота кэш промахов, то выражение (7.15) принимает вид:

$$T_p = (n^2 / p)(2n - 1) \cdot t + g(2 \cdot n^3/q + n^2)(a + 64/b) \quad (7.16)$$

#### 7.5.5. Программная реализация

Согласно вычислительной схеме блочного алгоритма умножения матриц, описанной в разделе 7.5.2, на каждой итерации алгоритма каждый поток параллельной программы выполняет вычисления над матричными блоками. Номер блока, который должен обрабатываться потоком в данный момент, вычисляется на основании положения потока в «решетке потоков» и номера текущей итерации.

Для определения числа потоков, которые будут использоваться при выполнении операции матричного умножения, введем переменную *ThreadNum*. Установим число потоков в значение *ThreadNum* при помощи функции *omp\_set\_num\_threads*. Как следует из описания параллельного алгоритма, число потоков должно являться полным квадратом

для того, чтобы потоки можно было представить в виде двумерной квадратной решетки. Определим размер «решетки потоков» *GridSize* и размер матричного блока *BlockSize*.

Для определения идентификатора потока воспользуемся функцией *omp\_get\_thread\_num*, сохраним результат в переменную *ThreadID*. Чтобы определить положение потока в решетке потоков, введем переменные *RowIndex* и *ColIndex*. Номер строки потоков, в котором расположен данный поток, есть результат целочисленного деления идентификатора потока на размер решетки потоков. Номер столбца, в котором расположен поток, есть остаток от деления идентификатора потока на размер решетки потоков.

Для выполнения операции матричного умножения каждый поток вычисляет элементы блока результирующей матрицы. Для этого каждый поток должен выполнить *GridSize* итераций алгоритма, каждая такая итерация есть умножение матричных блоков, итерации выполняются внешним циклом *for* по переменной *iter*.

```
// Программа 7.5
// Function for parallel calculating matrix multiplication
void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
    double* pCMatrix, int Size) {
    int ThreadNum = 4;
    int GridSize = int (sqrt((double)ThreadNum));
    int BlockSize = Size/GridSize;
    omp_set_num_threads(ThreadNum);
#pragma omp parallel
{
    int ThreadID = omp_get_thread_num();
    int RowIndex = ThreadID/GridSize;
    int ColIndex = ThreadID%GridSize;
    for (int iter=0; iter<GridSize; iter++) {
        for (int i=RowIndex*BlockSize; i<(RowIndex+1)*BlockSize; i++)
            for (int j=ColIndex*BlockSize; j<(ColIndex+1)*BlockSize; j++)
                for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
                    pCMatrix[i*Size+j] += pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
    }
} // pragma omp parallel
}
```

### 7.5.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при блочном разделении данных проводились при условиях, указанных в п. 7.3.6. Результаты вычислительных экспериментов приведены в таблице 7.8. Времена выполнения алгоритма указаны в секундах.

**Таблица 7.8.** Результаты вычислительных экспериментов для параллельного блочного алгоритма умножения матриц

Размер матриц	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
1000	24,82	4,40	5,64
1500	85,89	19,01	4,52
2000	176,58	44,90	3,93
2500	403,24	100,93	4,00
3000	707,15	172,88	4,09

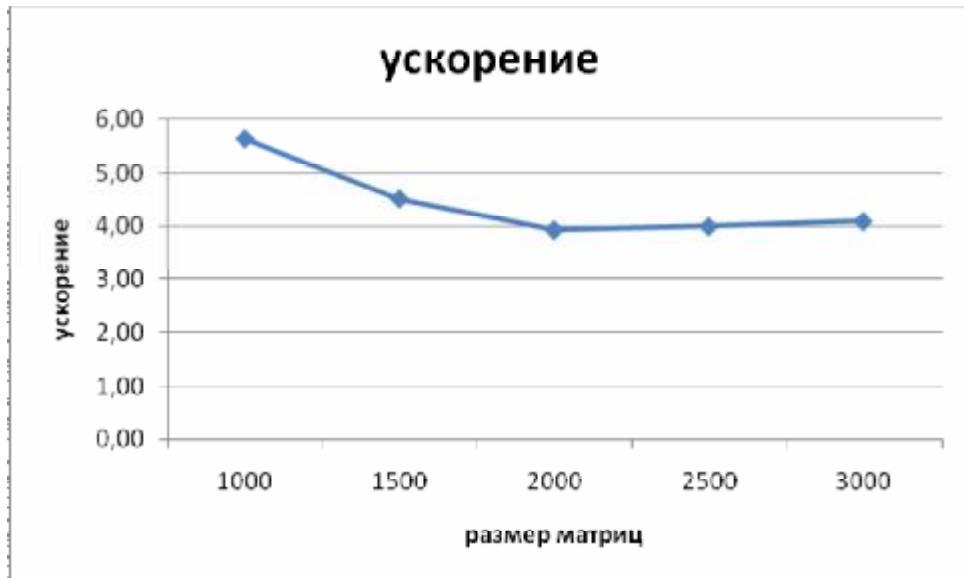


Рис. 7.12. Зависимость ускорения от количества исходных данных при выполнении параллельного блочного алгоритма умножения матриц

В таблице 7.9 и на рис. 7.13 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матриц с использованием четырех потоков со временем  $T_p^*$ , полученным при помощи модели (7.16). Частота кэш промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,24.

**Таблица 7.9.** Сравнение экспериментального и теоретического времени выполнения параллельного блочного алгоритма умножения матриц с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 7.15 – оценка сверху		Модель 7.16 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	4,3990	3,1994	13,1145	16,3139	3,1475	6,3469
1500	19,0077	10,7998	44,2466	55,0464	10,6192	21,4190
2000	44,8985	25,6016	104,8634	130,4650	25,1672	50,7688
2500	100,9302	50,0056	204,7908	254,7964	49,1498	99,1554
3000	172,8754	86,4126	353,8549	440,2675	84,9252	171,3378

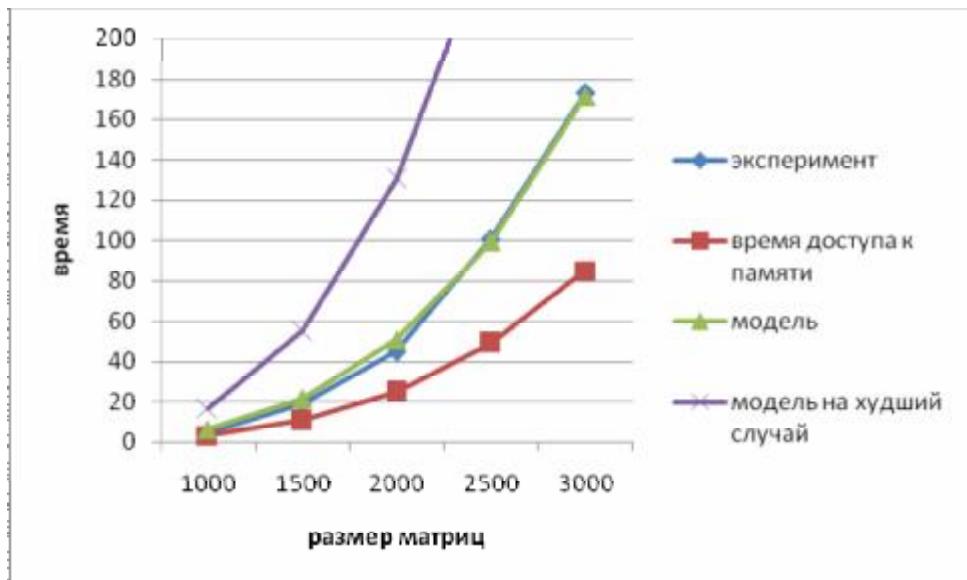


Рис. 7.13. График зависимости экспериментального и теоретического времени выполнения параллельного блочного алгоритма от объема исходных данных при использовании четырех потоков

## 7.6. Блочный алгоритм, эффективно использующий кэш-память

Как видно из результатов анализа эффективности рассмотренных параллельных алгоритмов, значительная доля времени умножения матриц тратится на многократное чтение элементов матриц  $A$  и  $B$  из оперативной памяти в кэш. Действительно, при вычислении элементов одной строки результирующей матрицы используются элементы одной строки матрицы  $A$  и все элементы матрицы  $B$  (рис. 7.1). Для вычисления первого элемента строки результирующей матрицы необходимо прочитать в кэш все элементы первой строки матрицы  $A$  и первого столбца матрицы  $B$ , при этом, поскольку матрица  $B$  хранится в памяти построчно, то элементы одного столбца располагаются в оперативной памяти с некоторым интервалом, чтение одного элемента столбца из оперативной памяти в кэш приводит к считыванию целой линейки данных размером 64 байта. Таким образом, можно сказать, что в кэш считывается не один столбец матрицы  $B$ , а сразу восемь столбцов. Полоса матрицы  $B$  может быть настолько велика, что чтение «последних» элементов этой полосы ведет к вытеснению из кэш «первых» элементов строки матрицы  $A$ . При вычислении второго элемента первой строки результирующей матрицы может произойти повторное чтение строки матрицы  $A$  и чтение второго столбца матрицы  $B$ . Так как размер матрицы  $B$  может быть настолько большим, что матрица не может быть помещена в кэш процессора полностью, то при вычислении последних элементов строки результирующей матрицы и чтении в кэш элементов последних столбцов, элементы первых столбцов матрицы  $B$  будут вытеснены из кэша. Далее, при вычислении элементов следующей строки результирующей матрицы элементы первых столбцов матрицы  $B$  необходимо будет снова загрузить в кэш. Итак, если размер матриц составляет  $n \times n$  элементов, то матрицы  $A$  и  $B$  могут переписываться  $n$  раз. Эта ситуация имеет место и в случае ленточного и блочного разбиения, если части матриц  $A$  и  $B$  не могут быть помещены в кэш полностью. Такая организация работы с кэш не может быть признана эффективной.

### 7.6.1. Последовательный алгоритм

Для организации алгоритма умножения матриц, который более эффективно использует кэш-память, воспользуемся разделением матриц на блоки. В случае, когда необходимо посчитать результат матричного умножения  $C = A \times B$ , и матрицы,

участвующие в умножении, настолько велики, что не могут быть помещены в кэш полностью, то становится возможным разделить матрицы на несколько матричных блоков меньшего размера и воспользоваться идеей блочного умножения матриц для того, чтобы получить результат матричного умножения. Разделение на блоки следует проводить таким образом, чтобы размер блока был настолько мал, что три блока, участвующие в умножении на данной итерации, возможно было поместить в кэш. Если блоки матриц  $A$  и  $B$  могут быть помещены в кэш полностью, то при вычислении результата умножения матричных блоков не происходит многократного чтения элементов блока в кэш, и, следовательно, затраты на загрузку данных из оперативной памяти существенно сокращаются.

**Программная реализация.** Для реализации последовательного алгоритма умножения матриц, основанного на блочном разбиении матриц и ориентированного на максимально эффективное использование кэш, воспользуемся алгоритмом, описанным в разделе 7.5. Отличие состоит лишь в том, что количество разбиений матриц теперь определяется не количеством потоков, а объемом кэш памяти.

Количество разбиений матриц должно быть таким, чтобы в кэш одновременно могли быть помещены три матричных блока – блоки матриц  $A$ ,  $B$  и  $C$ . Пусть  $V_{cache}$  – объем кэш в байтах. Тогда количество элементов типа double (числа с двойной точностью), которые могут быть помещены в кэш, составляет  $V/8$  (для хранения одного элемента типа double используется 8 байт). Таким образом, максимальный размер квадратного  $k \times k$  матричного блока составляет:

$$k_{\max} = \left\lfloor \sqrt{V_{cache} / (3 \cdot 8)} \right\rfloor.$$

Следовательно, минимально необходимое число разбиений  $GridSize$  при разделении матриц размером  $n \times n$  составляет:

$$GridSize = \lceil n / k_{\max} \rceil.$$

Для снижения сложности программной реализации алгоритма количество блоков по горизонтали и вертикали  $GridSize$  должно быть таким, чтобы размер матриц  $n$  мог быть поделен на  $GridSize$  без остатка.

В представленной программной реализации положим размер матричного блока равным 250. При таком размере блока три матричных блока могут быть одновременно помещены в кэш (для проведения экспериментов используется процессор Intel Core 2 Duo, объем кэш второго уровня составляет 2 Мб). Кроме того, такие размеры блоков в наших экспериментах гарантируют, что матрицы могут быть поделены на блоки поровну.

Рассмотрим программную реализацию последовательного блочного алгоритма матричного умножения.

```
// Программа 7.6
// Serial block matrix multiplication
void SerialResultCalculation (double* pAMatrix, double* pBMatrix,
    double* pCMatrix, int Size) {
    int BlockSize = 250;
    int GridSize = int (Size,double(BlockSize));
    for (int n=0; n<GridSize; n++)
        for (int m=0; m<GridSize; m++)
            for (int iter=0; iter<GridSize; iter++)
                for (int i=n*BlockSize; i<(n+1)*BlockSize; i++)
                    for (int j=m*BlockSize; j<(m+1)*BlockSize; j++)
                        for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
                            pCMatrix[i*Size+j] += pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
}
```

Следует отметить, что при использовании различных вычислительных систем с размером кэша, отличным от 2 Мб, следует варьировать параметр  $BlockSize$  таким образом, чтобы три матричных блока могли быть одновременно помещены в кэш.

### 7.6.2. Параллельный алгоритм

Для распараллеливания представленного блочного алгоритма умножения матриц воспользуемся подходом, который основывается на предложениях, изложенных при рассмотрении ленточного и блочного алгоритмов. Пусть, как и ранее, базовая подзадача (поток) отвечает за вычисление блока результирующей матрицы. Однако теперь, когда количество блоков определяется не количеством потоков, а объемом кэш памяти, число блоков может существенно превосходить число доступных потоков. Проведем агрегацию вычислений – пусть каждый поток обеспечивает получение нескольких матричных блоков результирующей матрицы  $C$ .

**1. Анализ эффективности.** При анализе эффективности параллельного алгоритма умножения матриц, основанного на разделении данных на блоки определенного размера с тем, чтобы максимально эффективно использовать кэш, можно использовать оценки, полученные при анализе предыдущего алгоритма. Действительно, объем вычислительных операций не изменяется. Поэтому вычислительная сложность блочного алгоритма составляет:

$$T_{\text{calc}} = (n^2 / p)(2n - 1) \cdot t.$$

При умножении матриц с помощью рассмотренного алгоритма выполняется  $q^3$  операций умножения матричных блоков (количество блоков по вертикали и горизонтали  $q$  определяется как результат деления порядка матриц  $n$  на количество строк и столбцов матричного блока  $k$ ). При этом, размер блоков определяется таким образом, чтобы на каждой итерации они могли быть одновременно помещены в кэш память. Значит, время, необходимое на чтение данных из оперативной памяти в кэш может быть вычислено по формуле:

$$T_{\text{mem}} = (n / k)^3 \cdot 3 \cdot k^2 (a + 64 / b).$$

Следовательно, общее время выполнения параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш, может быть вычислено по формуле:

$$T_p = (n^2 / p)(2n - 1) \cdot t + (n / k)^3 \cdot 3 \cdot k^2 (a + 64 / b). \quad (7.17)$$

Если дополнительно учесть частоту кэш промахов, то выражение (7.17) приобретает следующий вид:

$$T_p = (n^2 / p)(2n - 1) \cdot t + g(n / k)^3 \cdot 3 \cdot k^2 (a + 64 / b). \quad (7.18)$$

**2. Программная реализация.** Распределим между потоками параллельной программы итерации внешнего цикла (цикла по переменной  $n$ ). При таком распределении нагрузки на каждой итерации внешнего цикла поток последовательно выполняет поблочное умножение горизонтальной полосы матрицы  $A$  на вертикальные полосы матрицы  $B$ .

Программная реализация описанного подхода может быть получена следующим образом:

```
// Программа 7.7
// Parallel block matrix multiplication
void ParallelResultCalculation (double* pAMatrix, double* pBMatrix,
    double* pCMatrix, int Size) {
    int BlockSize = 250;
    int GridSize = int (Size / double(BlockSize));
#pragma omp parallel for
    for (int n=0; n<GridSize; n++)
        for (int m=0; m<GridSize; m++)
            for (int iter=0; iter<GridSize; iter++)
                for (int i=n*BlockSize; i<(n+1)*BlockSize; i++)
                    for (int j=m*BlockSize; j<(m+1)*BlockSize; j++)
```

```

        for (int k=iter*BlockSize; k<(iter+1)*BlockSize; k++)
            pCMatrix[i*Size+j] += pAMatrix[i*Size+k] * pBMatrix[k*Size+j];
    }
}

```

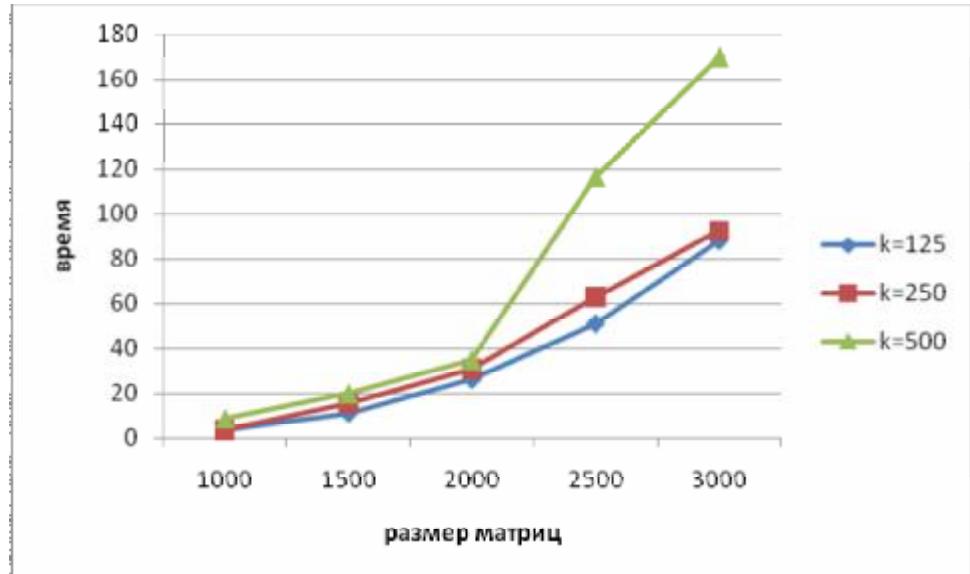
Можно отметить, что, несмотря на определенную логическую сложность рассмотренного алгоритма, его программная реализация не требует каких-либо значительных усилий.

### 7.6.3. Результаты вычислительных экспериментов

Вначале необходимо оценить, какое влияние оказывает выбор размера матричного блока (параметр  $k$ ) на эффективность параллельного алгоритма. Результаты вычислительных экспериментов, проведенных для параллельного алгоритма умножения матриц, эффективно использующего кэш память, при разном размере матричных блоков, приведены в таблице 7.10 и на рис. 7.14 (параметр  $k$  выбирался таким образом, чтобы матрицы могли быть поделены на равные блоки указанного размера).

**Таблица 7.10.** Время выполнения параллельного блочного алгоритма умножения матриц при разных размерах матричных блоков

Размер матриц	Параллельный блочный алгоритм умножения матриц, эффективно использующий кэш-память		
	$k=125$	$k=250$	$k=500$
1000	3,2505	3,3834	8,8142
1500	10,9575	15,4551	20,0040
2000	26,1422	31,0478	35,0169
2500	50,8252	62,8199	116,5353
3000	88,0399	92,4756	169,8120



**Рис. 7.14.** Время выполнения параллельного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, при разных размерах матричных блоков

Как видно из представленных результатов, если размер матричных блоков достаточно мал ( $k=125$  и  $k=250$ ) и все блоки, участвующие в умножении, могут быть одновременно помещены в кэш, то время выполнения алгоритма практически одинаково. Отсюда можно сделать вывод о том, что накладные расходы на чтение данных (последнее слагаемое в модели (7.18) тем больше, чем меньше  $k$ ) не оказывают существенного влияния на эффективность параллельного алгоритма. В случае, когда размеры матричных блоков

настолько велики, что все блоки, участвующие в умножении, не могут быть одновременно помещены в кэш, эффективность алгоритма существенно снижается, время выполнения алгоритма становится равным времени выполнения параллельных алгоритмов, описанных в предыдущих разделах.

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матриц при блочном разделении данных, ориентированного на эффективное использование кэш, проводились при условиях, указанных в п. 7.3.6. Результаты вычислительных экспериментов приведены в таблице 7.10. Времена выполнения алгоритма указаны в секундах.

**Таблица 7.11.** Результаты вычислительных экспериментов для параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти

Размер матрицы $p$	Базовый последовательный алгоритм ( $T_1$ )	Блочный последовательный алгоритм ( $T'_1$ )	Параллельный алгоритм		
			$T_p$	Ускорение	
				$T'_1 / T_p$	$T_1 / T_p$
1000	24,8192	13,7694	3,3834	4,0696	7,3355
1500	85,8869	47,1803	15,4551	3,0527	5,5572
2000	176,5772	126,6689	31,0478	4,0798	5,6873
2500	403,2405	213,0448	62,8199	3,3914	6,4190
3000	707,1501	377,2822	92,4756	4,0798	7,6469

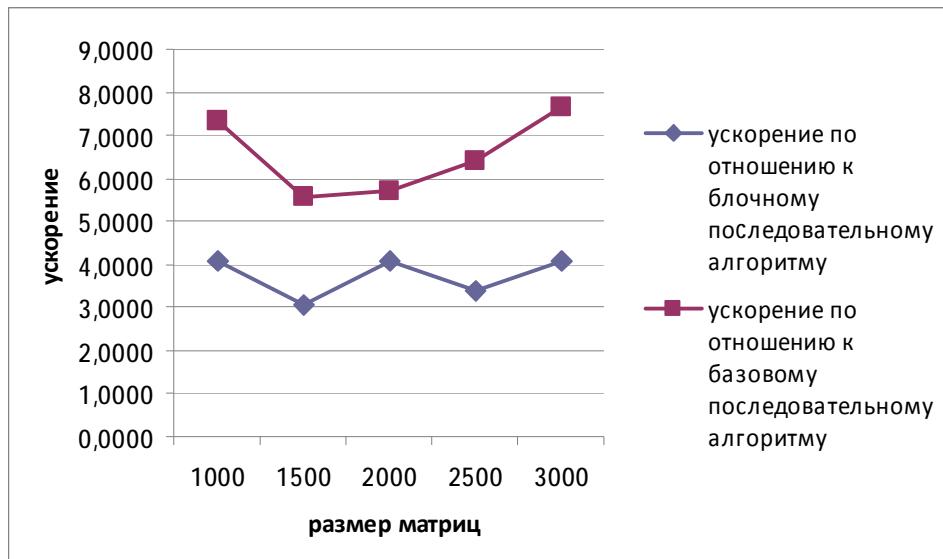


Рис. 7.15. Ускорение параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, в зависимости от размера матриц

В таблице 7.12 и на рис. 7.16 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма умножения матриц с использованием четырех потоков со временем  $T_p^*$ , полученным при помощи модели (7.18). Частота кэш промахов, измеренная с помощью системы VPS, для четырех потоков значение этой величины была оценена как 0,0026.

**Таблица 7.12.** Сравнение экспериментального и теоретического времени выполнения параллельного блочного алгоритма умножения матриц, ориентированного на эффективное использование кэш-памяти, с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 7.15 – оценка сверху		Модель 7.16 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	3,3834	3,1994	0,1572	3,3566	0,0004	3,1998
1500	15,4551	10,7998	0,5306	11,3304	0,0014	10,8012
2000	31,0478	25,6016	1,2577	26,8593	0,0033	25,6049
2500	62,8199	50,0056	2,4565	52,4621	0,0064	50,0120
3000	92,4756	86,4126	4,2448	90,6574	0,0110	86,4236

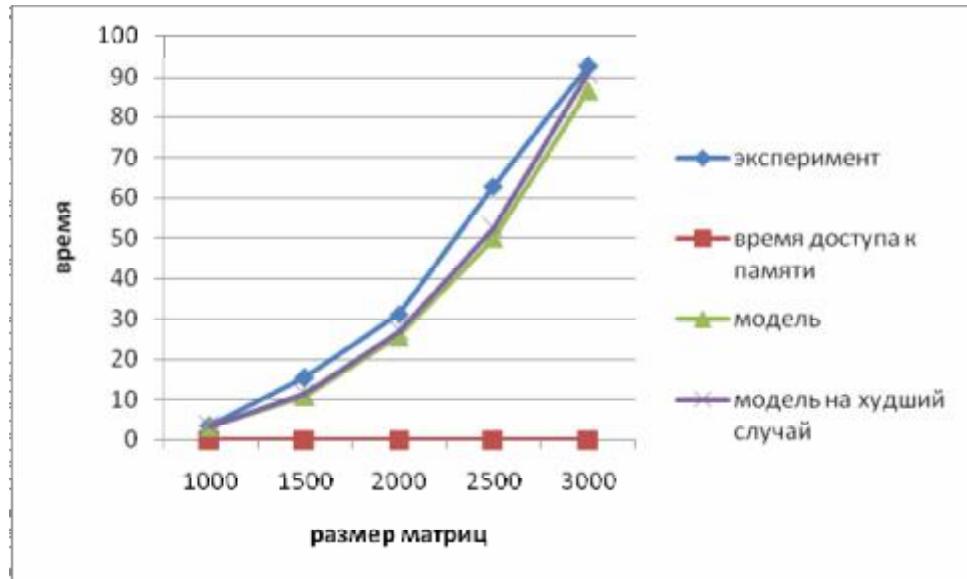


Рис. 7.16. График зависимости экспериментального и теоретического времени выполнения параллельного блочного алгоритма, ориентированного на эффективное использование кэш-памяти, от объема исходных данных при использовании четырех потоков

## 7.7. Краткий обзор главы

В данной главе рассмотрены четыре параллельных метода для выполнения операции матричного умножения. Первый и второй алгоритмы основаны на ленточном разделении матриц между процессорами. Первый вариант алгоритма основан на разделении между потоками параллельной программы одной из матриц-аргументов (матрицы  $A$ ) и матрицы-результата. Второй алгоритм основан на разделении первой матрицы на горизонтальные полосы, а второй матрицы – на вертикальные полосы, каждый поток параллельной программы в этом случае вычисляет один блок результирующей матрицы  $C$ . При реализации данного подхода используется механизм вложенного параллелизма. Также в разделе рассматриваются два блочных алгоритма умножения матриц, последний из которых основывается на разбиении матриц на блоки такого размера, чтобы блоки можно было полностью поместить в кэш-память.

На рис. 7.17 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов (ускорение параллельных алгоритмов показано относительно исходного последовательного метода умножения матриц).

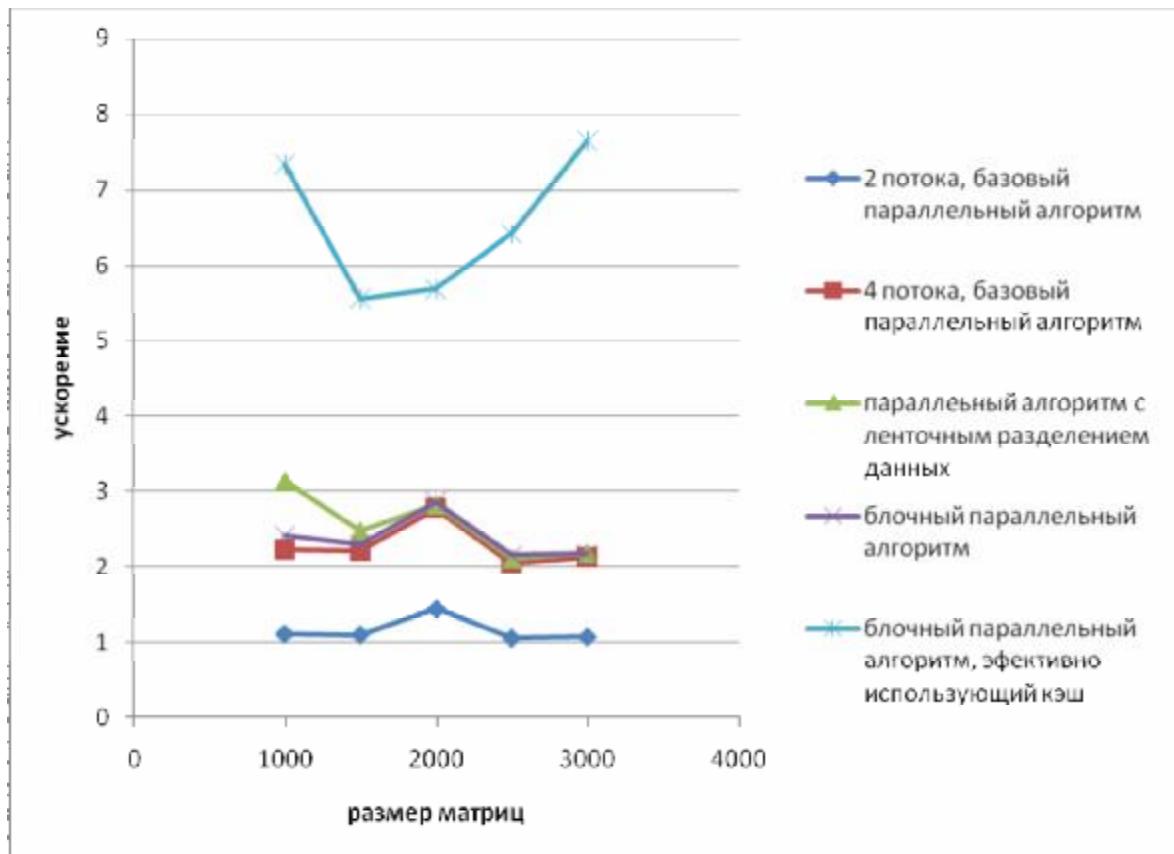


Рис. 7.17. Ускорение вычислений при матричном умножении для всех четырех рассмотренных в разделе параллельных алгоритмов

## 7.8. Обзор литературы

Задача умножения матриц широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы [8, 10, 72, 85]. Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе [50].

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа [45]. В данной работе рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

## 7.9. Контрольные вопросы

1. В чем состоит постановка задачи умножения матриц?
2. Приведите примеры задач, в которых используется операция умножения матриц.
3. Приведите примеры различных последовательных алгоритмов выполнения операции умножения матриц. Отличается ли их вычислительная трудоемкость?
4. Какие способы разделения данных используются при разработке параллельных алгоритмов матричного умножения?
5. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матриц.
6. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных?
7. Какие информационные взаимодействия выполняются для блочных алгоритмов умножения матриц?

8. Какой из рассмотренных алгоритмов обладает наилучшими показателями ускорения и эффективности?
9. Оцените возможность выполнения матричного умножения как последовательности операций умножения матрицы на вектор.
10. Какие функции библиотеки OpenMP оказались необходимыми при программной реализации параллельных алгоритмов?

### **7.10. Задачи и упражнения**

1. Выполните реализацию блочных алгоритмов умножения матриц, которые могли бы быть выполнены для прямоугольных решеток потоков общего вида.
2. Выполните реализацию матричного умножения с использованием ранее разработанных программ умножения матрицы на вектор.

Глава 8 . Параллельные методы решения систем линейных уравнений .....	1
8.1. Постановка задачи .....	1
8.2. Метод Гаусса .....	2
8.2.1. Общая схема метода .....	2
8.2.2. Прямой ход метода Гаусса .....	3
8.2.3. Обратный ход метода Гаусса .....	4
8.2.4. Программная реализация .....	5
8.2.5. Анализ эффективности .....	7
8.2.6. Результаты вычислительных экспериментов .....	9
8.3. Параллельный вариант метода Гаусса .....	10
8.3.1. Выделение информационных зависимостей .....	10
8.3.2. Масштабирование и распределение подзадач .....	10
8.3.3. Программная реализация .....	11
8.3.4. Анализ эффективности .....	14
8.3.5. Результаты вычислительных экспериментов .....	15
8.4. Метод сопряженных градиентов .....	18
8.4.1. Последовательный алгоритм .....	18
8.4.2. Организация параллельных вычислений .....	20
8.4.3. Программная реализация .....	20
8.4.4. Анализ эффективности .....	22
8.4.5. Результаты вычислительных экспериментов .....	23
8.5. Краткий обзор главы .....	26
8.6. Обзор литературы .....	26
8.7. Контрольные вопросы .....	27
8.8. Задачи и упражнения .....	27

## Глава 8 . Параллельные методы решения систем линейных уравнений

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых дифференциальными, интегральными или системами нелинейных (трансцендентных) уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

Матрицы коэффициентов систем линейных уравнений могут иметь различную структуру и свойства. Матрицы решаемых систем могут быть плотными и их порядок может достигать несколько тысяч строк и столбцов. При решении многих задач могут появляться системы, обладающие симметричными положительно определёнными ленточными матрицами с порядком в десятки тысяч и шириной ленты в несколько тысяч элементов. И, наконец, при рассмотрении большого ряда задач могут возникать системы линейных уравнений с разрежёнными матрицами с порядком в миллионы строк и столбцов.

### 8.1. Постановка задачи

*Линейное уравнение с n неизвестными*  $x_0, x_1, \dots, x_{n-1}$  может быть определено при помощи выражения

$$a_0x_0 + a_1x_1 + \dots + a_{n-1}x_{n-1} = b \quad (8.1)$$

где величины  $a_0, a_1, \dots, a_{n-1}$  и  $b$  представляют собой постоянные значения.

Множество  $n$  линейных уравнений

$$\begin{array}{lclcl} a_{0,0}x_0 & +a_{0,1}x_1 & +\dots+a_{0,n-1}x_{n-1} & =b_0 \\ a_{1,0}x_0 & +a_{1,1}x_1 & +\dots+a_{1,n-1}x_{n-1} & =b_1 \\ \dots & & & \\ a_{n-1,0}x_0 & +a_{n-1,1}x_1 & +\dots+a_{n-1,n-1}x_{n-1} & =b_{n-1} \end{array} \quad (8.2)$$

называется *системой линейных уравнений* или *линейной системой*. В более кратком (*матричном*) виде система может представлена как

$$Ax = b,$$

где  $A=(a_{i,j})$  есть вещественная матрица размера  $n \times n$ , а вектора  $b$  и  $x$  состоят из  $n$  элементов.

Под *задачей решения системы линейных уравнений* для заданных матрицы  $A$  и вектора  $b$  обычно понимается нахождение значения вектора неизвестных  $x$ , при котором выполняются все уравнения системы.

## 8.2. Метод Гаусса

Метод Гаусса является широко известным *прямым* алгоритмом решения систем линейных уравнений, для которых матрицы коэффициентов являются *плотными*. Если система линейных уравнений является *невырожденной*, то метод Гаусса гарантирует нахождение решения с погрешностью, определяемой точностью машинных вычислений. Основная идея метода состоит в приведении матрицы  $A$  посредством эквивалентных преобразований (не меняющих решение системы (8.2)) к треугольному виду, после чего значения искомых неизвестных может быть получено непосредственно в явном виде.

В подразделе дается общая характеристика метода Гаусса, достаточная для начального понимания алгоритма и позволяющая рассмотреть возможные способы параллельных вычислений при решении систем линейных уравнений. Более полное изложение алгоритма со строгим обсуждением вопросов точности получаемых решений может быть получено, например, в работах [2-4, 44, 68] и др.

### 8.2.1. Общая схема метода

Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решение рассматриваемой системы (такие преобразования носят наименование *эквивалентных*). К числу таких преобразований относятся:

- Умножение любого из уравнений на ненулевую константу,
- Перестановка уравнений,
- Прибавление к уравнению любого другого уравнения системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – *прямой ход* метода Гаусса – исходная система линейных уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду

$$Ux = c,$$

где матрица коэффициентов получаемой системы имеет вид

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ \dots & & & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}. \quad (8.3)$$

На *обратном ходе* метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы

может быть вычислено значение переменной  $x_{n-1}$ , после этого из предпоследнего уравнения становится возможным определение переменной  $x_{n-2}$  и т.д.

### 8.2.2. Прямой ход метода Гаусса

Прямой ход метода Гаусса состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений. На итерации  $i$ ,  $0 \leq i < n-1$ , метода производится исключение неизвестной  $i$  для всех уравнений с номерами  $k$ , больших  $i$  (т.е.  $i < k \leq n-1$ ). Для этого из этих уравнений осуществляется вычитание строки  $i$ , умноженной на константу  $(a_{ki}/a_{ii})$  с тем, чтобы результирующий коэффициент при неизвестной  $x_i$  в строках оказался нулевым – все необходимые вычисления могут быть определены при помощи соотношений:

$$\begin{aligned} a'_{kj} &= a_{kj} - (a_{ki}/a_{ii}) \cdot a_{ij}, & i \leq j \leq n-1, i < k \leq n-1, 0 \leq i < n-1 \\ b'_k &= b_k - (a_{ki}/a_{ii}) \cdot b_i, \end{aligned}$$

(следует отметить, что аналогичные вычисления выполняются и над вектором  $b$ ).

Поясним выполнение прямого хода метода Гаусса на примере системы линейных уравнений вида:

$$\begin{array}{rcl} x_0 & +3x_1 & +2x_2 = 1 \\ 2x_0 & +7x_1 & +5x_2 = 18 \\ x_0 & +4x_1 & +6x_2 = 26 \end{array}$$

На первой итерации производится исключение неизвестной  $x_0$  из второй и третьей строки. Для этого из этих строк нужно вычесть первую строку, умноженную соответственно на 2 и 1. После этих преобразований система уравнений принимает вид:

$$\begin{array}{rcl} x_0 & +3x_1 & +2x_2 = 1 \\ x_1 & +x_2 & = 16 \\ x_1 & +4x_2 & = 25 \end{array}$$

В результате остается выполнить последнюю итерацию и исключить неизвестную  $x_1$  из третьего уравнения. Для этого необходимо вычесть вторую строку и в окончательной форме система имеет следующий вид:

$$\begin{array}{rcl} x_0 & +3x_1 & +2x_2 = 1 \\ x_1 & +x_2 & = 16 \\ 3x_2 & & = 9 \end{array}$$

На рис. 8.1 представлена общая схема состояния данных на  $i$ -ой итерации прямого хода алгоритма Гаусса. Все коэффициенты при неизвестных, расположенные ниже главной диагонали и левее столбца  $i$ , уже являются нулевыми. На  $i$ -ой итерации прямого хода метода Гаусса осуществляется обнуление коэффициентов столбца  $i$ , расположенных ниже главной диагонали, путем вычитания строки  $i$ , умноженной на нужную ненулевую константу. После проведения  $(n-1)$  подобной итерации матрица, определяющая систему линейных уравнений, становится приведенной к верхнему треугольному виду.

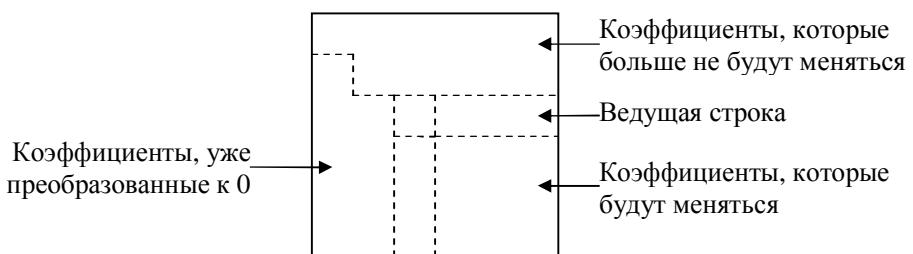


Рис. 8.1. Итерация прямого хода алгоритма Гаусса

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование *ведущей*, а диагональный элемент ведущей строки называется *ведущим элементом*. Как можно заметить, выполнение вычислений является возможным только, если ведущий элемент имеет ненулевое значение. Более того, если ведущий элемент  $a_{ii}$  имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Возможный способ избежать подобной проблемы может состоять в следующем – при выполнении каждой очередной итерации прямого хода метода Гаусса следует определить коэффициент с максимальным значением по абсолютной величине в столбце, соответствующем исключаемой неизвестной, т.е.

$$y = \max_{i \leq k \leq n-1} |a_{ki}|,$$

и выбрать в качестве ведущей строку, в которой этот коэффициент располагается (данная схема выбора ведущего значения носит наименование *метода главных элементов*).

Вычислительная сложность прямого хода алгоритма Гаусса с выбором ведущей строки имеет порядок  $O(n^3)$ .

### 8.2.3. Обратный ход метода Гаусса

После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной  $x_{n-1}$ , после этого из предпоследнего уравнения становится возможным определение переменной  $x_{n-2}$  и т.д. В общем виде, выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$\begin{aligned} x_{n-1} &= b_{n-1} / a_{n-1,n-1}, \\ x_i &= (b_i - \sum_{j=i+1}^{n-1} a_{ij}x_j) / a_{ii}, \quad i = n-2, n-3, \dots, 0 \end{aligned} \tag{8.4}$$

Поясним, как и ранее, выполнение обратного хода метода Гаусса на примере рассмотренной в п. 8.2.2 системы линейных уравнений

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 + x_2 &= 16. \\ 3x_2 &= 9 \end{aligned}$$

Из последнего уравнения системы можно определить, что неизвестная  $x_2$  имеет значение 3. В результате становится возможным разрешение второго уравнения и определение значение неизвестной  $x_1=13$ , т.е.

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 &= 13. \\ x_2 &= 3 \end{aligned}$$

На последней итерации обратного хода метода Гаусса определяется значение неизвестной  $x_0$ , равное -44.

С учетом последующего параллельного выполнения можно отметить, что учет получаемых значений неизвестных может выполняться сразу во всех уравнениях системы (и эти действия могут выполняться в уравнениях одновременно и независимо друг от друга). Так, в рассматриваемом примере после определения значения неизвестной  $x_2$  система уравнений может быть приведена к виду

$$\begin{array}{rcl}
 x_0 & +3x_1 & = -5 \\
 x_1 & & = 13 \\
 x_2 & & = 3
 \end{array}$$

Вычислительная сложность обратного хода алгоритма Гаусса составляет  $O(n^2)$ .

#### 8.2.4. Программная реализация

Рассмотрим возможный вариант реализации последовательного алгоритма Гаусса для решения систем линейных уравнений. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимании общей схемы вычислений.

**1. Главная функция программы.** Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```

// Программа 9.1 - Алгоритм Гаусса решения систем линейных уравнений
int* pPivotPos; // The number of pivot rows selected at the iterations
int* pPivotIter; // The iterations, at which the rows were pivots

void main() {
    double* pMatrix; // The matrix of the linear system
    double* pVector; // The right parts of the linear system
    double* pResult; // The result vector
    int Size; // The sizes of the initial matrix and the vector

    // Memory allocation and data initialization
    ProcessInitialization(pMatrix, pVector, pResult, Size);

    // Execution of Gauss algorithm
    SerialResultCalculation(pMatrix, pVector, pResult, Size);

    // Computational process termination
    ProcessTermination(pMatrix, pVector, pResult);
}

```

Следует пояснить использование дополнительных массивов. Элементы массива *pPivotPos* определяют номера строк матрицы, выбираемых в качестве ведущих, по итерациям прямого хода метода Гаусса. Именно в этом порядке должны выполняться затем итерации обратного хода для определения значений неизвестных системы линейных уравнений.

Элементы массива *pPivotIter* определяют номера итераций прямого хода метода Гаусса, на которых строки использовались в качестве ведущих (т.е., строка *i* выбиралась ведущей на итерации *pPivotIter[i]*). Начальное значение элементов массива устанавливается равным -1 и, тем самым, значение элемента массива *pPivotIter[i]*, равное -1, является признаком того, что строка *i* процесса все еще подлежит обработке. Кроме того, важно отметить, что запоминаемый в элементах массива *pPivotIter* номера итераций дополнительно означают и номера неизвестных, для определения которых будут использованы соответствующие строки уравнения.

Функция *ProcessInitialization* определяет исходные данные решаемой задачи (число неизвестных), выделяет память для хранения данных, осуществляет ввод матрицы коэффициентов системы линейных уравнений и вектора правых частей (или формирует эти данные при помощи датчика случайных чисел).

Функция *ProcessTermination* выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

**2. Функция SerialResultCalculation.** Реализует логику работы алгоритма Гаусса, последовательно вызывает функции, выполняющие прямой и обратный ход метода.

```
// Function for the execution of the Gauss algorithm
void SerialResultCalculation (double* pMatrix, double* pVector,
    double* pResult, int Size) {
    // Gaussian elimination
    SerialGaussianElimination (pMatrix, pVector, Size);
    // Back substitution
    SerialBackSubstitution (pMatrix, pVector, pResult, Size);
}
```

**3. Функция SerialGaussianElimination.** Функция выполняет прямой ход алгоритма Гаусса.

```
// Function for serial Gaussian elimination
void SerialGaussianElimination (double* pMatrix, double* pVector, int Size) {
    int Iter;           // The number of the iteration
    int PivotRow;      // The number of the current pivot row
    for (Iter=0; Iter<Size; Iter++) {
        // Finding the pivot row
        PivotRow = FindPivotRow(pMatrix, Size, Iter);
        pPivotPos[Iter] = PivotRow;
        pPivotIter[PivotRow] = Iter;
        SerialColumnElimination(pMatrix, pVector, Size, Iter, PivotRow);
    }
}
```

**4. Функция FindPivotRow.** Функция *FindPivotRow* определяет строку линейной системы, которую следует использовать в качестве ведущей на данной итерации алгоритма. В качестве ведущей строки выбирается строка, которая ранее не использовалась в качестве ведущей (т.е. для которой элемент массива *pSerialPivotIter* равен -1) с максимальным по абсолютному значению элементом, расположенным в столбце *Iter*.

```
// Function for finding pivot row
int FindPivotRow (double* pMatrix, int Size, int Iter) {
    int PivotRow = -1;      // The index of the pivot row
    double MaxValue = 0;   // The value of the pivot element
    int i;                 // Loop variable

    // Choose the row, that stores the maximum element
    for (i=0; i<Size; i++) {
        if ((pSerialPivotIter[i] == -1) &&
            (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
            PivotRow = i;
            MaxValue = fabs(pMatrix[i*Size+Iter]);
        }
    }
    return PivotRow;
}
```

**5. Функция SerialColumnElimination.** Функция *SerialColumnElimination* проводит вычитание ведущей строки из строк процесса, которые еще не использовались в качестве ведущих (т.е. для которых элементы массива *pSerialPivotIter* равны -1).

```
// Function for column elimination
void SerialColumnElimination (double* pMatrix, double* pVector, int Size,
    int Iter, int Pivot) {
    double PivotValue, PivotFactor;
    PivotValue = pMatrix[Pivot*Size+Iter];
    for (int i=0; i<Size; i++) {
        if (pSerialPivotIter[i] == -1) {
            PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
            for (int j=Iter; j<Size; j++) {
```

```

    pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
}
pVector[i] -= PivotFactor * pVector[Pivot];
}
}
}
```

**6. Функция SerialBackSubstitution.** Функция реализует обратный ход метода Гаусса.

```
// Function for serial back substitution
void SerialBackSubstitution (double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int RowIndex, Row;
    for (int i=Size-1; i>=0; i--) {
        RowIndex = pPivotPos[i];
        pResult[i] = pVector[RowIndex]/pMatrix[RowIndex*Size+i];
        pMatrix[RowIndex*Size + i] = 1;
        for (int j=0; j<i; j++) {
            Row = pPivotPos[j];
            pVector[Row] -= pMatrix[Row*Size+i]*pResult[i];
            pMatrix[Row*Size+i] = 0;
        }
    }
}
```

### **8.2.5. Анализ эффективности**

При анализе эффективности последовательного алгоритма умножения матриц снова используем подход, примененный в п. 6.5.4.

Итак, время выполнения алгоритма складывается из времени, которое тратится непосредственно на вычисления, и времени, необходимого на чтение данных из оперативной памяти в кэш процессора.

Оценим количество вычислительных операций, выполняемых на каждой итерации прямого и обратного хода алгоритма Гаусса.

На каждой итерации прямого хода метода Гаусса необходимо осуществить выбор ведущей строки среди строк, подлежащих обработке. На  $i$ -ой итерации прямого хода число строк, подлежащих обработке, равно  $(n-i)$ , где  $n$  – порядок матрицы линейной системы. После выбора ведущей строки, производится вычитание этой строки, умноженной на константу, из всех строк, подлежащих обработке. Каждая строка содержит  $(n-i)$  ненулевых элементов, число строк –  $(n-i)$ . Таким образом, количество вычислительных операций, необходимых для вычитания ведущей строки из остальных строк составляет  $2(n-i) \cdot (n-i)$ . Для приведения матрицы к верхне-треугольному виду необходимо выполнить  $(n-1)$  итерацию прямого хода метода Гаусса. Следовательно, время выполнения вычислений составляет:

$$T_{calc}^1 = \sum_{i=0}^{n-2} [(n-i) + 2 \cdot (n-i)^2] \cdot t .$$

На каждой итерации обратного хода метода Гаусса во всех строках линейной системы, подлежащих обработке, выполняется корректировка элемента вектора правых частей. На  $i$ -ой итерации обратного хода число строк, подлежащих обработке, равно  $(n-i)$ . Для вычисления значений всех неизвестных необходимо выполнить  $(n-1)$  итерацию обратного хода. Следовательно, количество вычислительных операций может быть получено по формуле:

$$T_{calc}^2 = \sum_{i=0}^{n-2} 2 \cdot (n-i) \cdot t .$$

Итак, время, которое тратится непосредственно на вычисления при выполнении последовательного алгоритма Гаусса:

$$T_{calc} = \sum_{i=0}^{n-2} [(n-i) + 2 \cdot (n-i)^2 + 2(n-i)] \cdot t = \frac{4n^3 + 15n^2 + 11n - 30}{6} t. \quad (8.5)$$

Теперь необходимо оценить объем данных, которые нужно прочитать из оперативной памяти в кэш вычислительного элемента в случае, когда размер линейной системы настолько велик, что матрица и вектора, описывающие эту систему, одновременно не могут быть помещены в кэш.

На каждой итерации прямого хода метода Гаусса для выбора ведущей строки необходимо просмотреть элементы, расположенные в  $i$ -ом столбце у всех строк, подлежащих обработке, и выбрать среди них максимальный по абсолютному значению элемент. Соответственно, все эти элементы необходимо прочитать из оперативной памяти. Количество строк, подлежащих обработке, на  $i$ -ой итерации прямого хода составляет  $(n-i)$ . Напомним также, что считывание данных происходит не по одному элементу, а кэш-строками по 64 байта. Следовательно, из оперативной памяти считывается  $64 \cdot (n-i)$  байт. После выбора ведущей строки для выполнения вычитания необходимо прочитать из оперативной памяти ненулевые элементы строк, подлежащих обработке и соответствующие элементы вектора правых частей. Количество строк, подлежащих обработке, составляет  $(n-i)$ , в каждой строке содержится  $(n-i)$  ненулевых элементов. Следовательно, объем загружаемых данных  $64 \cdot (n-i)(n-i) + 64 \cdot (n-i)$ .

Таким образом, время, которое тратится на считывание необходимых данных из оперативной памяти в кэш при выполнении прямого хода метода Гаусса, составляет:

$$T_{mem}^1 = \frac{64 \cdot \sum_{i=0}^{n-2} [(n-i) + (n-i)^2 + (n-i)]}{b} = \frac{64 \cdot \sum_{i=0}^{n-2} [(n-i)^2 + 2 \cdot (n-i)]}{b}.$$

На каждой  $i$ -ой итерации обратного хода метода Гаусса производится зануление элементов, расположенных в  $i$ -ом столбце и корректировка элементов вектора правых частей в строках линейной системы, подлежащих обработке. Число таких строк равно  $(n-i)$ . Для выполнения вычислений необходимо прочитать из памяти элементы матрицы и вектора, подлежащие изменению. Подлежащие обработке элементы матрицы расположены в памяти не последовательно, следовательно считывание из оперативной памяти в кэш будет происходить линейками по 64 байта. Таким образом, время, необходимое на чтение необходимых данных при выполнении обратного хода метода Гаусса может быть определено по формуле:

$$T_{mem}^2 = \frac{64 \cdot \sum_{i=0}^{n-2} [(n-i) + (n-i)]}{b} = \frac{64 \cdot 2 \cdot \sum_{i=0}^{n-2} (n-i)}{b}.$$

Общее время, необходимое на считывание необходимых данных из оперативной памяти в кэш при выполнении последовательного алгоритма Гаусса составляет:

$$T_{mem} = \frac{64 \cdot \sum_{i=0}^{n-2} [(n-i)^2 + 2 \cdot (n-i)]}{b} + \frac{64 \cdot 2 \cdot \sum_{i=0}^{n-2} (n-i)}{b} = \frac{64 \cdot (2n^3 + 15n^2 + 13n - 30)}{6b}. \quad (8.6)$$

Следовательно, общее время выполнения последовательного алгоритма Гаусса составляет:

$$T_1 = \frac{4n^3 + 15n^2 + 11n - 30}{6} t + \frac{64 \cdot (2n^3 + 15n^2 + 13n - 30)}{6b}. \quad (8.7)$$

Если, как и в предыдущих разделах, учесть латентность памяти, то модель приобретет следующий вид:

$$T_1 = \frac{4n^3 + 15n^2 + 11n - 30}{6} t + \frac{(2n^3 + 15n^2 + 13n - 30)}{6} \left( a + \frac{64}{b} \right). \quad (8.8)$$

Полученная модель является моделью на худший случай. Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см п. 6.5.4):

$$T_1 = \frac{4n^3 + 15n^2 + 11n - 30}{6} t + g \frac{(2n^3 + 15n^2 + 13n - 30)}{6} \left( a + \frac{64}{b} \right). \quad (8.9)$$

### 8.2.6. Результаты вычислительных экспериментов

Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1,86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Для снижения сложности построения теоретических оценок времени выполнения метода Гаусса, при компиляции и построения программ для проведения вычислительных экспериментов функция оптимизации кода компилятором была отключена (результаты оценки влияния компиляторной оптимизации на эффективность программного кода приведены в п. 7.2.4).

Для того, чтобы оценить время одной операции  $\tau$ , измерим время выполнения последовательного алгоритма Гаусса при малых объемах данных, таких, чтобы матрица линейной системы, вектор правых частей и вектор-результат полностью поместились в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицу линейной системы и вектор правых частей случайными числами (для обеспечения гарантированной разрешимости системы уравнений исходная матрица задается в виде нижней треугольной матрицы), а вектор-результат – нулями. Выполнение этих действий гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение  $\tau$ , равное 5,820 нс.

Оценка величины пропускной способности канала доступа к оперативной памяти  $\beta$  проводилась в п. 6.5.4 и определена для используемого вычислительного узла как 12,44 Гб/с, а латентность памяти  $a = 8,31$  нс.

В таблице 8.1 и на рис. 8.2 представлены результаты сравнения времени выполнения  $T_1$  последовательного алгоритма Гаусса со временем  $T_1^*$ , полученным при помощи модели (8.9). Частота кэш промахов, измеренная с помощью системы VPS, для одного потока значение этой величины была оценена как 0,0133.

Таблица 8.1. Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма Гаусса

Размер матриц	$T_1$	$T_1^*$ (calc) (модель)	Модель 8.8 – оценка сверху		Модель 8.9 – уточненная оценка	
			$T_1^*$ (mem)	$T_1^*$	$T_1^*$ (mem)	$T_1^*$
1000	3,7488	3,8946	4,3999	8,2945	0,0585	3,9531
1500	12,9136	13,1278	14,8128	27,9405	0,1970	13,3248
2000	30,5710	31,0982	35,0681	66,1663	0,4664	31,5646
2500	59,6408	60,7160	68,4411	129,1570	0,9103	61,6262
3000	103,0359	104,8910	118,2072	223,0982	1,5722	106,4631

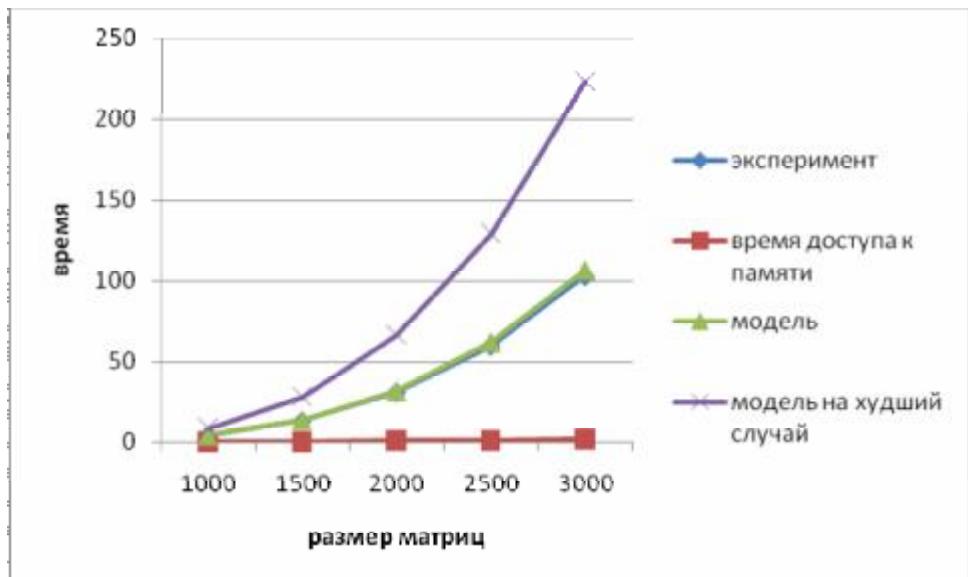


Рис. 8.2. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных

### 8.3. Параллельный вариант метода Гаусса

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве базовой подзадачи можно принять тогда все вычисления, связанные с обработкой одной строки матрицы  $A$  и соответствующего элемента вектора  $b$ .

#### 8.3.1. Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить  $(n-1)$  итерацию по исключению неизвестных для преобразования матрицы коэффициентов  $A$  к верхнему треугольному виду.

Выполнение итерации  $i$ ,  $0 \leq i < n-1$ , прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца  $i$ , соответствующего исключаемой переменной  $x_i$ . Зная ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной  $x_i$ .

При выполнении **обратного хода** метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача  $i$ ,  $0 \leq i < n-1$ , определяет значение своей переменной  $x_i$ , это значение должно быть использовано всеми подзадачами с номерами  $k$ ,  $k < i$ , для выполнения корректировки значений элементов вектора  $b$ .

#### 8.3.2. Масштабирование и распределение подзадач

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью. Вместе с этим, следует учитывать, что по мере выполнения вычислений

часть подзадач будет завершать свои вычисления (например, после того, как строка подзадачи использовалась в качестве ведущей при прямом ходе метода Гаусса). Решение этой проблемы может состоять, например, в укрупнении подзадач – тем более, что обычно размер матрицы, описывающей систему линейных уравнений, оказывается большим, чем число доступных вычислительных элементов (т.е.,  $p < n$ ). При этом для объединения нескольких строк в рамках единой подзадачи целесообразно использовать не последовательную, а циклическую схему разделения матрицы на горизонтальные полосы (см. подраздел 6.2).

### 8.3.3. Программная реализация

Проведя анализ последовательного варианта алгоритма Гаусса, можно заключить, что распараллеливание возможно для следующих вычислительных процедур:

- поиск ведущей строки (функция *FindPivotRow*),
- вычитание ведущей строки из всех строк, подлежащих обработке (функция *EliminateColumns*),
- выполнение обратного хода метода Гаусса.

**1. Распараллеливание процедуры поиска ведущей строки.** На каждой  $i$ -ой итерации прямого хода метода Гаусса необходимо просмотреть строки, подлежащие обработке, и выбрать среди них строку, у которой в  $i$ -ом столбце расположен максимальный по абсолютному значению элемент. Каждый поток параллельной программы отвечает за вычисления, производимые над множеством строк матрицы линейной системы и соответствующими элементами векторов. Можно изменить алгоритм таким образом, чтобы сначала все потоки параллельно определили «локальную» ведущую строку в рамках своей полосы строк матрицы, а затем осуществить выбор «глобальной» ведущей строки среди «локальных» ведущих строк потоков.

Для этого объявим новую структуру данных для хранения «локальной» ведущей строки *TThreadPivotRow*. Полями этой структуры являются номер «локальной» ведущей строки в матрице линейной системы и значение, расположенное в столбце с исключаемой неизвестной.

Для того, чтобы выбор ведущей строки выполнялся потоками параллельно, объявим параллельный фрагмент при помощи директивы *parallel*. Внутри параллельного фрагмента объявим переменную типа *TThreadPivotRow* для хранения «локальной» ведущей строки. Поскольку данная переменная объявлена внутри параллельного фрагмента, локальные копии этой переменной будут созданы во всех потоках параллельной секции. Для распределения итераций основного цикла выбора ведущей строки между потоками воспользуемся директивой *for*.

После того, как все потоки определили «локальную» ведущую строку, до завершения параллельного фрагмента необходимо выполнить редукцию полученных значений и выбрать «глобальную» ведущую строку. Для этого необходимо просмотреть выбранные «локальные» ведущие строки и выбрать среди них строку с максимальным значением поля *MaxValue*. Выбор глобальной ведущей строки можно обеспечить при помощи механизма *критических секций* (директива *critical*). Код, расположенный внутри критической секции, в каждый момент времени выполняется только одним потоком (любые потоки, пытающиеся получить доступ к уже выполняемой критической секции, блокируются). После того, как выполнение критической секции завершается, к ее выполнению может приступить другой поток (в первую очередь из числа блокированных при доступе к данной критической секции). Подобная схема выполнения обычно именуется *взаимоисключением* – более подробно понятие критической секции рассмотрено в разделе 4 учебных материалов курса.

С учетом высказанных предложений, параллельный вариант функции *FindPivotRow* может быть представлен следующим образом.

```
// Программа 9.2 - Параллельный алгоритм Гаусса
// решения систем линейных уравнений

typedef struct {
    int PivotRow;
    double MaxValue;
} TThreadPivotRow;

// Finding the pivot row - parallel implementation
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
    int PivotRow = -1; // The index of the pivot row
    double MaxValue = 0; // The value of the pivot element
    int i; // Loop variable

    // Choose the row, that stores the maximum element
#pragma omp parallel
{
    TThreadPivotRow ThreadPivotRow;
    ThreadPivotRow.MaxValue = 0;
    ThreadPivotRow.PivotRow = -1;
#pragma omp for
    for (i=0; i<Size; i++) {
        if ((pSerialPivotIter[i] == -1) &&
            (fabs(pMatrix[i*Size+Iter]) > ThreadPivotRow.MaxValue)) {
            ThreadPivotRow.PivotRow = i;
            ThreadPivotRow.MaxValue = fabs(pMatrix[i*Size+Iter]);
        }
    }
#pragma omp critical
{
        if (ThreadPivotRow.MaxValue > MaxValue){
            MaxValue = ThreadPivotRow.MaxValue;
            PivotRow = ThreadPivotRow.PivotRow;
        }
    } // pragma omp critical
} // pragma omp parallel
    return PivotRow;
}
```

Следует понимать, что введение синхронизации при организации критической секции уменьшит возможность достижения максимально возможного ускорения параллельных вычислений. Для снижения нежелательных эффектов синхронизации можно, например, при помощи запоминания результатов потоков (параметров «потоковых» ведущих строк) в отдельном массиве, в котором далее выполнить обычный последовательный поиск (реализация данного варианта поиска ведущей строки может быть использована в качестве темы самостоятельного упражнения).

**2. Распараллеливание процедуры исключения очередной неизвестной.** После выбора «глобальной» ведущей строки необходимо занулить элементы столбца с исключаемой неизвестной во всех строках, подлежащих обработке. Реализуем параллельный вариант этого алгоритма в функции *ParallelEliminateColumns*.

За обработку одной строки матрицы линейной системы отвечает одна итерация внешнего цикла по переменной *i*. Поскольку преобразования строк осуществляются независимо, то можно распределить их между параллельными потоками и выполнить параллельно. Для этого воспользуемся директивой *parallel for* для распределения итераций внешнего цикла между потоками. При этом переменная *PivotFactor*, которая используется для хранения множителя, на который умножается текущая строка, должна быть локализована (параметр *private* директивы *parallel for*):

```

// Column elimination - parallel implementation
void ParallelColumnElimination (double* pMatrix, double* pVector, int Size,
    int Iter, int Pivot) {
    double PivotValue, PivotFactor;
    PivotValue = pMatrix[Pivot*Size+Iter];
#pragma omp parallel for private (PivotFactor)
    for (int i=0; i<Size; i++) {
        if (pSerialPivotIter[i] == -1) {
            PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
            for (int j=Iter; j<Size; j++) {
                pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
            }
            pVector[i] -= PivotFactor * pVector[Pivot];
        }
    }
}

```

Следует отметить, что по умолчанию итерации цикла распределяются между потоками поровну. Так, например, если для выполнения цикла, содержащего 100 итераций, используется два потока, то первый поток отвечает за выполнение итераций с номерами от 0 до 49, а второй – за выполнение итераций с номерами от 50 до 99. В случае, если итерации цикла имеют разный объем вычислений, такое способ распределения итераций может приводить к неравномерной вычислительной нагрузке потоков.

При выполнении прямого хода метода Гаусса после выбора ведущей строки вычисления производятся только над строками, подлежащими обработке (строка с номером  $i$  подлежит обработке, если она ранее не выбиралась в качестве ведущей – признаком такой ситуации является значение элемента массива  $pSerialPivotIter[i]$  равное -1). Расположение уже отработанных строк может быть достаточно произвольным и определяется порядком выбора ведущих строк в соответствии с методом главных элементов. Как результат, итерации цикла могут обладать разной вычислительной сложностью – итерация цикла может быть «нагруженной», если она соответствует еще неотработанной строке и необходимо выполнить вычитание строк, а может быть и вычислительно простой при попадании на уже отработанную строку матрицы.

Для того, чтобы обеспечить равномерную загрузку вычислительных элементов, можно изменить принятое правило по умолчанию для распределения итераций между потоками при помощи параметра *schedule* директивы *parallel for*, выбрав, например, динамическую схему распределения итераций между потоками. В этом случае, итерации цикла распределяются между потоками поблочно; при завершении обработки очередного блока каждый поток получает для вычислений очередной блок итераций – более подробно правила управления распределением итераций цикла между потоками рассмотрены в разделе 5 учебных материалов.

Как можно видеть, подобная динамическая схема распределения итераций цикла между потоками может решить проблему балансировки вычислений. Возможный улучшенный вариант параллельного выполнения процедуры исключения очередной неизвестной для прямого хода методы Гаусса может выглядеть следующим образом:

```

// Column elimination - parallel implementation
void ParallelColumnElimination (double* pMatrix, double* pVector, int Pivot,
    int Iter, int Size) {
    double PivotValue, PivotFactor;
    PivotValue = pMatrix[Pivot*Size+Iter];
#pragma omp parallel for private(PivotFactor) schedule(dynamic,1)
    for (int i=0; i<Size; i++) {
        ...
    }
}

```

Динамическая схема (*dynamic*) организует подобие «очереди итераций»: каждый поток берет на выполнение текущую итерацию из очереди и как только итерация выполнена, происходит новое обращение к очереди за новой итерацией. Такой способ распределения вычислительной нагрузки гарантирует балансировку вычислительных элементов.

Используя разработанные функции *ParallelFindPivotRow* и *ParallelColumnElimination* параллельный вариант прямого хода метода Гаусса может быть представлен в следующем виде:

```
// Gaussian elimination - parallel implementation
void ParallelGaussianElimination(double* pMatrix, double* pVector, int Size) {
    int Iter;           // The number of the iteration
    int PivotRow;       // The number of the current pivot row
    for (Iter=0; Iter<Size; Iter++) {
        // Finding the pivot row
        PivotRow = ParallelFindPivotRow(pMatrix, Size, Iter);
        pPivotPos[Iter] = PivotRow;
        pPivotIter[PivotRow] = Iter;
        ParallelColumnElimination(pMatrix, pVector, Size, Iter, PivotRow);
    }
}
```

**3. Распараллеливание обратного хода метода Гаусса.** Итерации обратного хода метода Гаусса должны выполняться строго последовательно. Однако обработка строк линейной системы на каждой итерации обратного хода происходит независимо, и, следовательно, данные вычисления можно распределить между параллельными потоками согласно вычислительной схеме параллельного алгоритма.

При помощи директивы *parallel for* распределим между потоками параллельной программы итерации внутреннего цикла, при этом переменная *Row*, которая хранит номер текущей обрабатываемой строки, должна быть локализована. Получаемый в результате параллельный вариант функции для выполнения обратного хода метода Гаусса может выглядеть следующим образом:

```
// Back substution - parallel implementation
void ParallelBackSubstitution (double* pMatrix, double* pVector,
                               double* pResult, int Size) {
    int RowIndex, Row;
    for (int i=Size-1; i>=0; i--) {
        RowIndex = pSerialPivotPos[i];
        pResult[i] = pVector[RowIndex]/pMatrix[RowIndex*Size+i];
        pMatrix[RowIndex*Size+i] = 1;
#pragma omp parallel for private (Row)
        for (int j=0; j<i; j++) {
            Row = pSerialPivotPos[j];
            pVector[Row] -= pMatrix[Row*Size+i]*pResult[i];
            pMatrix[Row*Size+i] = 0;
        }
    }
}
```

#### 8.3.4. Анализ эффективности

При разработке параллельного алгоритма все вычислительные операции, выполняемые алгоритмом Гаусса, были распределены между потоками параллельной программы. Однако, для организации параллельного метода главных элементов потребовалось ввести дополнительный блок кода, отвечающий за редукцию данных, полученных разными потоками.

Следовательно, время, необходимое для выполнения вычислений на этапе прямого хода, можно определить при помощи соотношения:

$$T_p(\text{calc}) = T_1(\text{calc}) / p + 3pt,$$

где второе слагаемое  $3pt$  определяет затраты на выбор «глобальной» ведущей строки. Подставив выражение  $T_1(\text{calc})$  получим, что время выполнения вычислений для параллельного варианта метода Гаусса описывается выражением:

$$T_p(\text{calc}) = [(4n^3 + 15n^2 + 11n - 30) / 6p]t + 3pt. \quad (8.10)$$

Поскольку обращения к памяти параллельными потоками осуществляются строго последовательно, то время на чтение необходимых данных из оперативной памяти в кэш при выполнении параллельного метода Гаусса совпадает со временем, полученным при анализе последовательного алгоритма:

$$T_p(\text{mem}) = \frac{64 \cdot (2n^3 + 15n^2 + 13n - 30)}{6b} \quad (8.11)$$

Если учесть латентность памяти:

$$T_p(\text{mem}) = \frac{(2n^3 + 15n^2 + 13n - 30)}{6} \left( a + \frac{64}{b} \right) \quad (8.12)$$

Теперь необходимо оценить величину накладных расходов, обусловленных организацией и закрытием параллельных секций. Как отмечалось в главе 6, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс. Параллельная секция создается при каждом выборе ведущей строки, при выполнении вычитания ведущей строки из остальных строк линейной системы, подлежащих обработке, а также при выполнении каждой итерации обратного хода метода Гаусса. Таким образом, общее число параллельных секций составляет  $3 \cdot (n-1)$ .

Сводя воедино все полученные оценки можно заключить, что время выполнения параллельного метода Гаусса описывается соотношением:

$$T_p = \frac{4n^3 + 15n^2 + 11n - 30}{6p} t + 3pt + \frac{(2n^3 + 15n^2 + 13n - 30)}{6} \left( a + \frac{64}{b} \right) + 3(n-1)d \quad (8.13)$$

Для уточнения модели следует учесть коэффициент кэш промахов  $\gamma$ :

$$T_p = \frac{4n^3 + 15n^2 + 11n - 30}{6p} t + 3pt + \gamma \frac{(2n^3 + 15n^2 + 13n - 30)}{6} \left( a + \frac{64}{b} \right) + 3(n-1)d \quad (8.14)$$

### 8.3.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода Гаусса для решения систем линейных уравнений проводились при условиях, указанных в п. 6.5.5 и состоят в следующем.

Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

Результаты вычислительных экспериментов приведены в таблице 8.2. Времена выполнения алгоритмов указаны в секундах.

Таблица 8.2. Результаты вычислительных экспериментов для параллельного алгоритма Гаусса

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
1000	3,7488	2,8875	1,2983	1,5627	2,3989

1500	12,9136	7,9223	1,6300	4,7501	2,7186
2000	30,5710	17,0476	1,7933	11,3656	2,6898
2500	59,6408	31,8271	1,8739	22,2635	2,6789
3000	103,0359	53,7655	1,9164	38,5342	2,6739



Рис. 8.3. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма Гаусса

В таблицах 8.3 и 8.4 и на рис. 8.4 и 8.5 представлены результаты сравнения времени выполнения  $T_p$  параллельного алгоритма Гаусса с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (8.14). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,045, а для четырех потоков значение этой величины была оценена как 0,077.

Таблица 8.3. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Гаусса с использованием двух потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 8.13 – оценка сверху		Модель 8.14 – уточненная оценка	
			$T_p^* (met)$	$T_p^*$	$T_p^* (met)$	$T_p^*$
1000	2,8875	1,9480	4,3999	6,3479	0,1980	2,1460
1500	7,9223	6,5650	14,8128	21,3778	0,6666	7,2316
2000	17,0476	15,5506	35,0681	50,6187	1,5781	17,1287
2500	31,8271	30,3599	68,4411	98,8009	3,0798	33,4397
3000	53,7655	52,4477	118,2072	170,6549	5,3193	57,7671

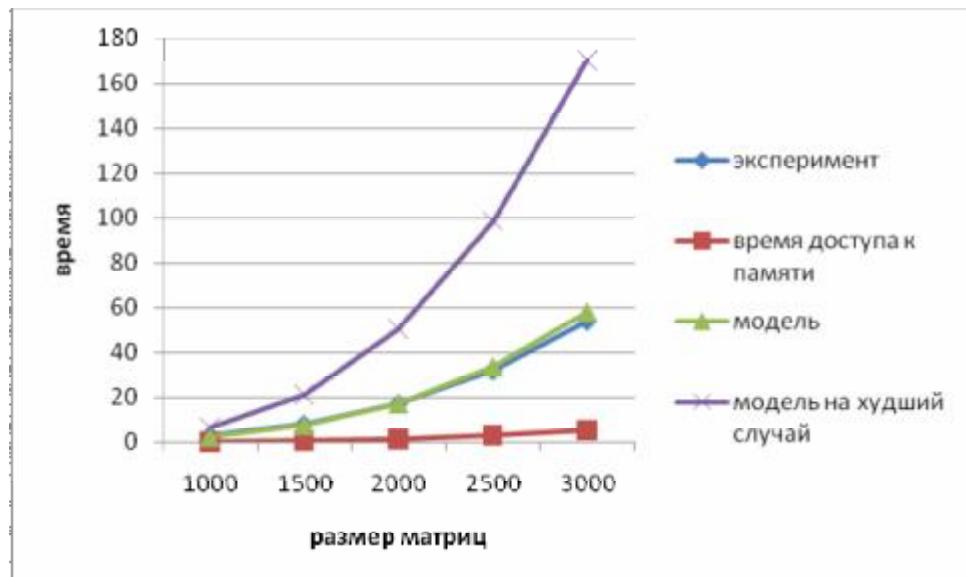


Рис. 8.4. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Гаусса от объема исходных данных при использовании двух потоков

Таблица 8.4. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Гаусса с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 8.13 – оценка сверху		Модель 8.14 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
1000	1,5627	0,9744	4,3999	5,3743	0,1980	1,1724
1500	4,7501	3,2831	14,8128	18,0958	0,6666	3,9496
2000	11,3656	7,7761	35,0681	42,8441	1,5781	9,3541
2500	22,2635	15,1809	68,4411	83,6219	3,0798	18,2607
3000	38,5342	26,2250	118,2072	144,4322	5,3193	31,5443

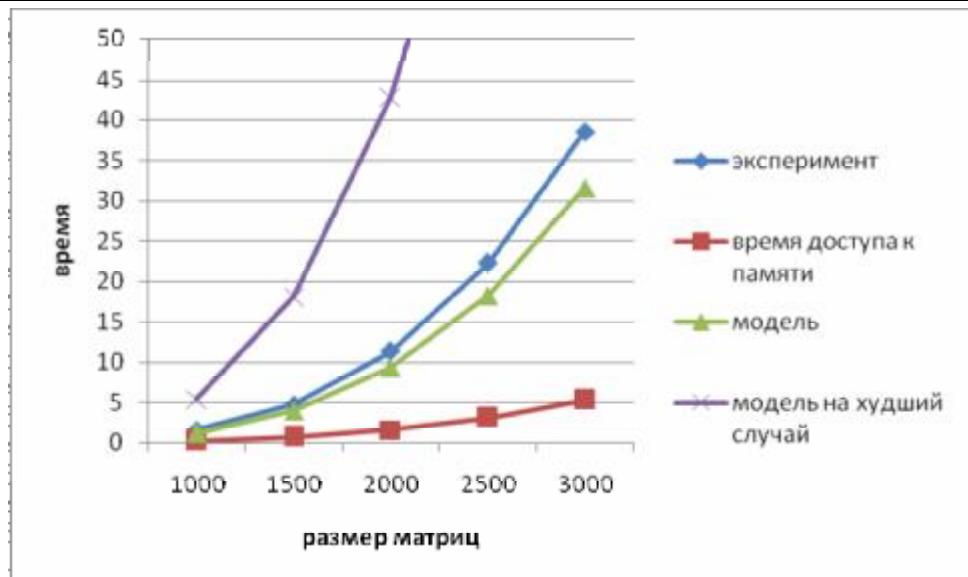


Рис. 8.5. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Гаусса от объема исходных данных при использовании четырех потоков

## 8.4. Метод сопряженных градиентов

Рассмотрим теперь совершенно иной подход к решению систем линейных уравнений, при котором к искомому точному решению  $x^*$  системы  $Ax=b$  строится последовательность приближенных решений  $x^0, x^1, \dots, x^k, \dots$ . При этом процесс вычислений организуется таким способом, что каждое очередное приближение дает оценку точного решения со все уменьшающейся погрешностью, и при продолжении расчетов оценка точного решения может быть получена с любой требуемой точностью. Подобные *итерационные методы* решения систем линейных уравнений широко используются в практике вычислений. К преимуществам итерационных методов можно отнести меньший объем (по сравнению, например, с методом Гаусса) необходимых вычислений для решения разреженных систем линейных уравнений, возможность более быстрого получения начальных приближений искомого решения, наличие эффективных способов организации параллельных вычислений. Дополнительная информация с описанием таких методов, рассмотрение вопросов сходимости и точности получаемых решений может быть получена, например, в [2-4,44].

Одним из наиболее известных итерационных алгоритмов является *метод сопряженных градиентов*, который может быть применен для решения симметричной положительно определенной системы линейных уравнений большой размерности.

Напомним, что матрица  $A$  является *симметричной*, если она совпадает со своей транспонированной матрицей, т.е.  $A=A^T$ . Матрица  $A$  называется *положительно определенной*, если для любого вектора  $x$  справедливо:  $x^T A x > 0$ .

Известно (см., например, [2-4,44]), что после выполнения  $n$  итераций алгоритма ( $n$  есть порядок решаемой системы линейных уравнений), очередное приближение  $x^n$  совпадает с точным решением.

### 8.4.1. Последовательный алгоритм

Если матрица  $A$  симметричная и положительно определенная, то функция

$$q(x) = \frac{1}{2} x^T \cdot A \cdot x - x^T b + c \quad (8.15)$$

имеет единственный минимум, который достигается в точке  $x^*$ , совпадающей с решением системы линейных уравнений (8.2). *Метод сопряженных градиентов* является одним из широкого класса итерационных алгоритмов, которые позволяют найти решение (8.2) путем минимизации функции  $q(x)$ .

*Итерация метода сопряженных градиентов* состоит в вычислении очередного приближения к точному решению в соответствии с правилом:

$$x^k = x^{k-1} + s^k d^k. \quad (8.16)$$

Тем самым, новое значение приближения  $x^k$  вычисляется с учетом приближения, построенного на предыдущем шаге  $x^{k-1}$ , скалярного шага  $s^k$  и вектора направления  $d^k$ .

Перед выполнением первой итерации вектора  $x^0$  и  $d^0$  полагаются равными нулю, а для вектора  $g^0$  устанавливается значение равное  $-b$ . Далее каждая итерация для вычисления очередного значения приближения  $x^k$  включает выполнение четырех шагов:

**Шаг 1:** Вычисление градиента:

$$g^k = A \cdot x^{k-1} - b; \quad (8.17)$$

**Шаг 2:** Вычисление вектора направления:

$$d^k = -g^k + \frac{(g^k)^T, g^k}{(g^{k-1})^T, g^{k-1}} d^{k-1}, \quad (8.18)$$

где  $(g^T, g)$  есть скалярное произведение векторов;

**Шаг 3:** Вычисление величины смещения по выбранному направлению:

$$s^k = \frac{(d^k, g^k)}{(d^k)^T \cdot A \cdot d^k}; \quad (8.19)$$

**Шаг 4:** Вычисление нового приближения:

$$x^k = x^{k-1} + s^k d^k. \quad (8.20)$$

Как можно заметить, данные выражения включают две операции умножения матрицы на вектор, четыре операции скалярного произведения и пять операций над векторами. Как результат, что общее количество числа операций, выполняемых на одной итерации, составляет

$$t_1 = 4n^2 + 11n.$$

Как уже отмечалось ранее, для нахождения точного решения системы линейных уравнений с положительно определенной симметричной матрицей необходимо выполнить  $n$  итераций. Таким образом, для нахождения решения системы необходимо выполнить

$$T_1 = 4n^3 + 11n^2, \quad (8.21)$$

и, тем самым, сложность алгоритма имеет порядок  $O(n^3)$ .

Поясним выполнение метода сопряженных градиентов на примере решения системы линейных уравнений вида:

$$\begin{aligned} 3x_0 - x_1 &= 3 \\ -x_0 + 3x_1 &= 7 \end{aligned}$$

Эта система уравнений второго порядка обладает симметричной положительно определенной матрицей, для нахождения точного решения этой системы достаточно провести всего две итерации метода.

На первой итерации было получено значение градиента  $g^1 = (-3, -7)$ , значение вектора направления  $d^1 = (3, 7)$ , значение величины смещения  $s^1 = 0.439$ . Соответственно, очередное приближение к точному решению системы  $x^1 = (1.318, 3.076)$ .

На второй итерации было получено значение градиента  $g^2 = (-2.121, 0.909)$ , значение вектора направления  $d^2 = (2.397, -0.266)$ , а величина смещения –  $s^2 = 0.284$ . Очередное приближение  $x^2 = (2, 3)$  совпадает с точным решением системы  $x^*$ .

На рис. 8.6 представлена последовательность приближений к точному решению, построенная методом сопряженных градиентов (в качестве начального приближения  $x^0$  выбрана точка  $(0,0)$ ).

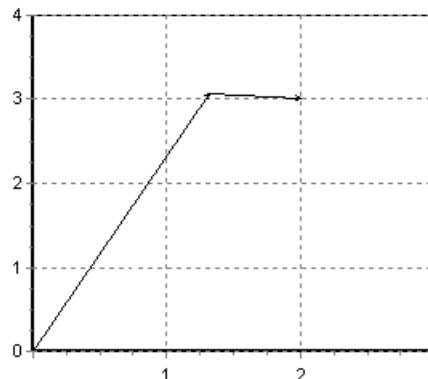


Рис. 8.6. Итерации метода сопряженных градиентов при решении системы второго порядка

### 8.4.2. Организация параллельных вычислений

При разработке параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений в первую очередь следует учесть, что выполнение итераций метода осуществляется последовательно и, тем самым, наиболее целесообразный подход состоит в распараллеливании вычислений, реализуемых в ходе выполнения итераций.

Анализ соотношений (8.17)-(8.21) показывает, что основные вычисления, выполняемые в соответствии с методом, состоят в умножении матрицы  $A$  на вектора  $x$  и  $d$ , и, как результат, при организации параллельных вычислений может быть полностью использован материал, изложенный в разделе 6.

Дополнительные вычисления в (8.17)-(8.21), имеющие меньший порядок сложности, представляют собой различные операции обработки векторов (скалярное произведение, сложение и вычитание, умножение на скаляр). Организация таких вычислений, конечно же, должна быть согласована с выбранным параллельным способом выполнения операции умножения матрицы на вектор.

### 8.4.3. Программная реализация

Рассмотрим возможный вариант реализации параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений с симметричной положительно-определенной матрицей. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимании общей схемы вычислений.

**1. Главная функция программы.** Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 9.3 - Метод сопряженных градиентов для
//                   решения систем линейных уравнений

void main() {
    double* pMatrix; // The matrix of the linear system
    double* pVector; // The right parts of the linear system
    double* pResult; // The result vector
    int Size;         // The sizes of the initial matrix and the vector

    printf ("Conjugate Gradient Method for Solving Linear Systems!\n");

    // Memory allocation and data initialization
    ProcessInitialization(pMatrix, pVector, pResult, Size);

    // Execution of Conjugate Gradient Method
    ParallelResultCalculation(pMatrix, pVector, pResult, Size);

    // Computational process termination
    ProcessTermination(pMatrix, pVector, pResult);
}
```

Функция *ProcessInitialization* определяет исходные данные решаемой задачи (число неизвестных), выделяет память для хранения данных, осуществляет ввод матрицы коэффициентов системы линейных уравнений и вектора правых частей (или формирует эти данные при помощи датчика случайных чисел).

Функция *ProcessTermination* выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

**2. Функция ParallelResultCalculation.** Реализует логику работы параллельного варианта метода сопряженных градиентов.

```
// Conjugate Gradient Method - parallel implementation
void ParallelResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    double * CurrentApproximation,      * PreviousApproximation;
    double * CurrentGradient,          * PreviousGradient;
    double * CurrentDirection,         * PreviousDirection;
    double * Denom;
    double Step;

    int Iter = 1, MaxIter = Size + 1;
    float Accuracy = 0.0001f;

    AllocateVectors(CurrentApproximation, PreviousApproximation,
        CurrentGradient, PreviousGradient, CurrentDirection, PreviousDirection,
        Denom, Size);

    for (int i=0; i<Size; i++) {
        PreviousApproximation[i] = 0;
        PreviousDirection[i] = 0;
        PreviousGradient[i] = -pVector[i];
    }

    do {
        if (Iter > 1) {
            SwapPointers(PreviousApproximation, CurrentApproximation);
            SwapPointers(PreviousGradient, CurrentGradient);
            SwapPointers(PreviousDirection, CurrentDirection);
        }
        //compute gradient
#pragma omp parallel for
        for (int i=0; i<Size; i++) {
            CurrentGradient[i] = -pVector[i];
            for (int j=0; j<Size; j++)
                CurrentGradient[i] += pMatrix[i*Size+j]*PreviousApproximation[j];
        }

        //compute direction
        double IP1 = 0, IP2 = 0;
#pragma omp parallel for reduction(+:IP1,IP2)
        for (int i=0; i<Size; i++) {
            IP1 += CurrentGradient[i]*CurrentGradient[i];
            IP2 += PreviousGradient[i]*PreviousGradient[i];
        }
#pragma omp parallel for
        for (i=0; i<Size; i++) {
            CurrentDirection[i] = -CurrentGradient[i] +
                PreviousDirection[i]*IP1/IP2;
        }

        //compute size step
        IP1 = 0;
        IP2 = 0;
#pragma omp parallel for reduction(+:IP1,IP2)
        for (int i=0; i<Size; i++) {
            Denom[i] = 0;
            for (int j=0; j<Size; j++)
                Denom[i] += pMatrix[i*Size+j]*CurrentDirection[j];
            IP1 += CurrentDirection[i]*CurrentGradient[i];
            IP2 += CurrentDirection[i]*Denom[i];
        }
        Step = -IP1/IP2;
```

```

    //compute new approximation
#pragma omp parallel for
for (i=0; i<Size; i++) {
    CurrentApproximation[i] = PreviousApproximation[i] +
        Step*CurrentDirection[i];
}
Iter++;
} while
((Dest(PreviousApproximation, CurrentApproximation, Size) > Accuracy)
&& (Iter < MaxIter));

for (int i=0; i<Size; i++)
pResult[i] = CurrentApproximation[i];

DeleteVectors(CurrentApproximation, PreviousApproximation, CurrentGradient,
    PreviousGradient, CurrentDirection, PreviousDirection, Denom);
}

```

#### 8.4.4. Анализ эффективности

Выберем для дальнейшего анализа эффективности получаемых параллельных вычислений параллельный алгоритм матрично-векторного умножения при ленточном горизонтальном разделении матрицы. При этом операции над векторами, обладающие меньшей вычислительной трудоемкостью, также будем выполнять в многопоточном режиме.

Вычислительная трудоемкость последовательного метода сопряженных градиентов была уже определена ранее в (8.21).

Определим время выполнения параллельной реализации метода сопряженных градиентов. Вычислительная сложность параллельных операций умножения матрицы на вектор при использовании схемы ленточного горизонтального разделения матрицы составляет (здесь и далее  $L$  – количество итераций, выполняемых методом):

$$T_p^1(\text{calc}) = L \cdot \frac{2n \cdot (2n-1)}{p} \cdot t \quad (\text{см. раздел 6}).$$

Все остальные операции над векторами (скалярное произведение, сложение, умножение на константу), также выполняются в многопоточном режиме. Следовательно, общая вычислительная сложность параллельного варианта метода сопряженных градиентов является равной:

$$T_p(\text{calc}) = L \cdot \frac{4n^2 + 11n}{p} \cdot t. \quad (8.22)$$

Уточним теперь приведенные выражения – оценим трудоемкость операции считывания необходимых данных из оперативной памяти в кэш вычислительных элементов. Пусть размер матрицы линейной системы настолько велик, что матрица и вектора, принимающие участие в вычислениях, не могут быть одновременно помещены в кэш. Отдельно рассмотрим каждый из 4 шагов, выполняемых на каждой итерации метода сопряженных градиентов.

В операции вычисления градиента принимают участие матрица линейной системы, вектор правых частей, вектор предыдущего приближения и, непосредственно, сам вектор градиента. Следовательно, объем данных, прочитанных из оперативной памяти, составляет:

$$T_p^1(\text{mem}) = \frac{64(n^2 + 3n)}{b}.$$

Для вычисления текущего вектора направления необходимо получить доступ к текущему и предыдущему векторам градиентов, к вектору направления, полученному на предыдущем шаге, а также к текущему вектору направления. Следовательно, время, необходимое на загрузку данных, составляет:

$$T_p^2(\text{mem}) = \frac{64 \cdot 4n}{b}.$$

Далее для вычисления величины смещения в выбранном направлении необходимо выполнить вычисления, в которых участвуют матрица линейной системы, а также вектор градиента и вектор направления. Затраты на загрузку необходимых данных:

$$T_p^3(\text{mem}) = \frac{64(n^2 + 2n)}{b}.$$

При вычислении очередного приближения используются вектора предыдущего и текущего приближений и вектор направления. Следовательно, время на чтение необходимых данных из оперативной памяти в кэш:

$$T_p^4(\text{mem}) = \frac{64 \cdot 3n}{b}.$$

Далее для определения достигнутой точности вычисляется расстояние между предыдущим и текущим приближением. Для этого необходимо иметь доступ к этим векторам:

$$T_p^5(\text{mem}) = \frac{64 \cdot 2n}{b}.$$

Следовательно, время, которое тратится на загрузку необходимых данных при выполнении параллельного варианта метода сопряженных градиентов, составляет:

$$T_p(\text{mem}) = L \cdot \frac{64 \cdot (2n^2 + 14n)}{b}. \quad (8.23)$$

В модели необходимо также учесть латентность оперативной памяти (см п. 6.5.4):

$$T_p(\text{mem}) = L \cdot (2n^2 + 14n) \cdot \left( a + \frac{64}{b} \right). \quad (8.24)$$

Суммируя полученные выражения как итог следует, что общее время выполнения алгоритма в худшем случае составляет:

$$T_p = L \cdot \frac{4n^2 + 11n}{p} \cdot t + L \cdot (2n^2 + 14n) \cdot \left( a + \frac{64}{b} \right) \quad (8.25)$$

Для построения точной модели необходимо также учесть коэффициент кэш промахов (см п. 6.5.4):

$$T_p = L \cdot \frac{4n^2 + 11n}{p} \cdot t + g \cdot L \cdot (2n^2 + 14n) \cdot \left( a + \frac{64}{b} \right) \quad (8.26)$$

#### 8.4.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений с симметричной положительно-определенной матрицей проводились при условиях, указанных в п. 8.3.5. Результаты вычислительных экспериментов приведены в таблице 8.5. Времена выполнения алгоритмов указаны в секундах.

Таблица 8.5. Результаты вычислительных экспериментов для параллельного метода сопряженных градиентов

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
1000	20,06335	11,24981	1,7834	5,631906	3,5624
1500	67,79017	38,71493	1,7510	19,57367	3,4633
2000	160,5922	92,15922	1,7426	46,41194	3,4601
2500	315,8357	180,7888	1,7470	91,32836	3,4582
3000	546,2592	312,0792	1,7504	157,7329	3,4632

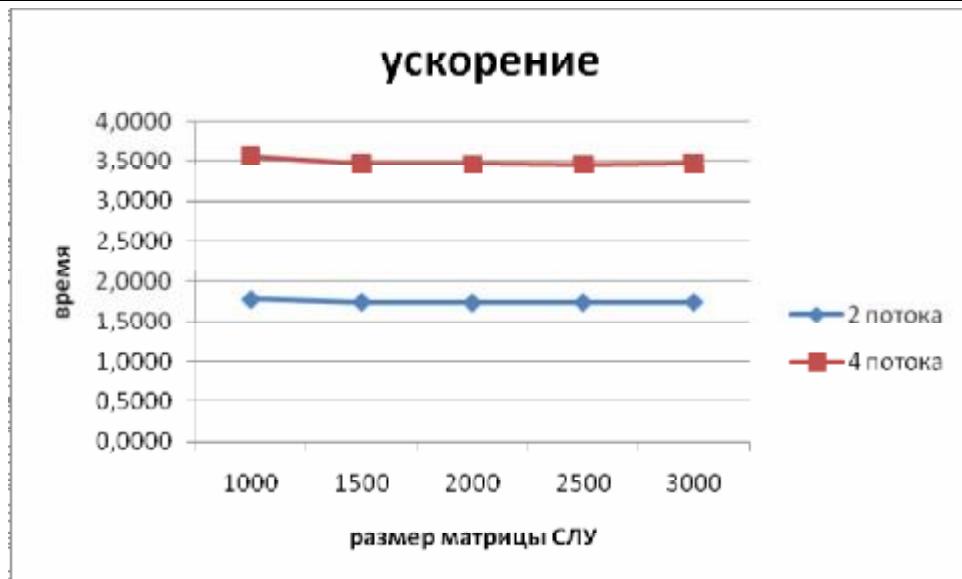


Рис. 8.7. Зависимость ускорения от количества исходных данных при выполнении параллельного метода сопряженных градиентов

В таблице 8.6 и 8.7 и на рис. 8.8 и 8.9 представлены результаты сравнения времени выполнения  $T_p$  параллельного метода сопряженных градиентов с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (8.26) (длительность одной базовой вычислительной операции  $\tau$  равна 5,057 нс). Количество проведенных итераций метода всегда равнялось количеству строк матрицы системы линейных уравнений. Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0085, а для четырех потоков значение этой величины была оценена как 0,0102.

Таблица 8.6. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов с использованием двух потоков

Размер матриц	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 8.25 – оценка сверху		Модель 8.26 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
1000	11,2498	10,1418	26,3862	36,5280	0,2243	10,3661
1500	38,7149	34,1973	88,8469	123,0443	0,7552	34,9525
2000	92,1592	81,0233	210,3556	291,3789	1,7880	82,8113
2500	180,7888	158,2051	410,5642	568,7693	3,4898	161,6949
3000	312,0792	273,3283	709,1248	982,4531	6,0276	279,3559

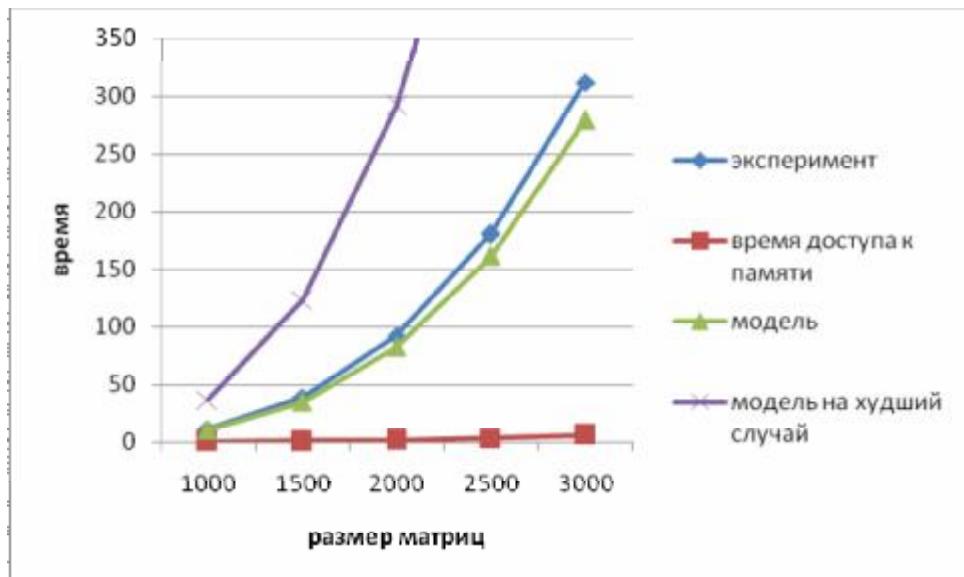


Рис. 8.8. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов от объема исходных данных при использовании двух потоков

Таблица 8.7. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 8.25 – оценка сверху		Модель 8.26 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
1000	5,6319	5,0709	26,3862	31,4571	0,2691	5,3400
1500	19,5737	17,0987	88,8469	105,9456	0,9062	18,0049
2000	46,4119	40,5116	210,3556	250,8672	2,1456	42,6573
2500	91,3284	79,1025	410,5642	489,6667	4,1878	83,2903
3000	157,7329	136,6642	709,1248	845,7889	7,2331	143,8972

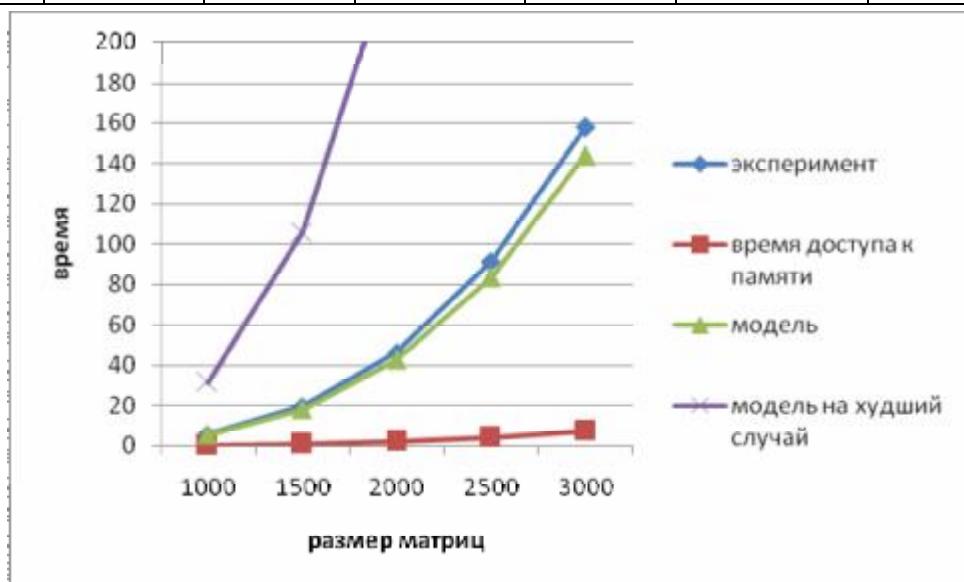


Рис. 8.9. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма сопряженных градиентов от объема исходных данных при использовании четырех потоков

## 8.5. Краткий обзор главы

Данная глава посвящена проблеме параллельных вычислений при решении систем линейных уравнений. Изложение учебного материала проводится с использованием двух широко известных алгоритма – *метод Гаусса* как пример прямого алгоритма решения задачи и итерационный *метод сопряженных градиентов*.

*Параллельный вариант метода Гаусса* основывается на ленточном разделении матрицы между вычислительными элементами. Для определения параллельного варианта метода проводится полный цикл проектирования – определяются базовые подзадачи, выделяются информационные взаимодействия, обсуждаются вопросы масштабирования, выводятся оценки показателей эффективности, предлагается схема программной реализации и приводятся результаты вычислительных экспериментов. При программной реализации алгоритма для лучшей балансировки вычислительной нагрузки между потоками используется поддерживаемая в OpenMP динамическая схема распределения вычислений.

Важный момент при рассмотрении *параллельного варианта метода сопряженных градиентов* состоит в том, что способ параллельных вычислений для этого метода может быть получен через параллельные алгоритмы выполняемых вычислительных действий – операций умножения матрицы на вектор, скалярного произведения векторов, сложения и вычитания векторов. Выбранный для учебного изучения подход состоит в распараллеливании всех операций над матрицами и векторами. Такой подход позволил организовать параллельные вычисления с достаточно высокими показателями эффективности - см. рис. 8.10.

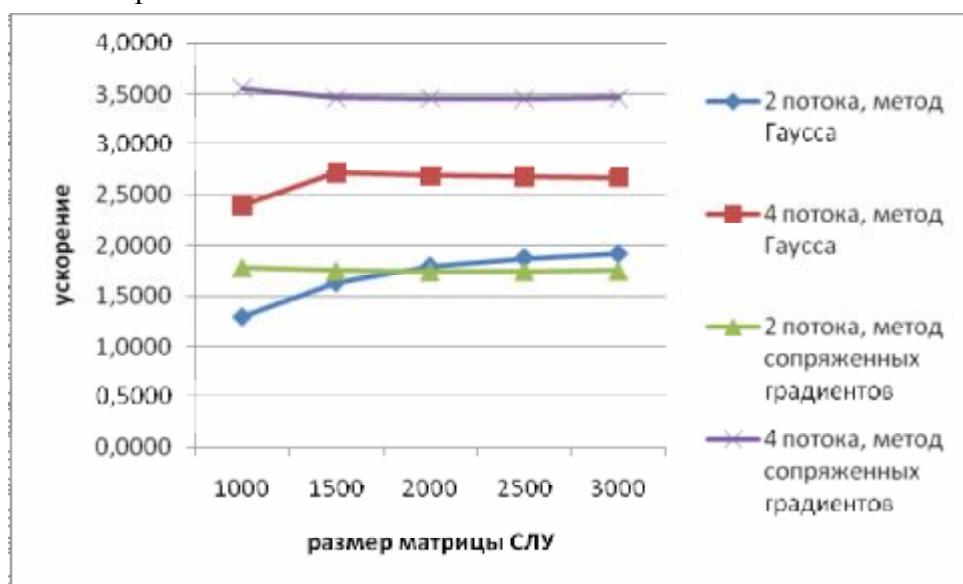


Рис. 8.10. Ускорение параллельных алгоритмов решения системы линейных уравнений в зависимости от размера матрицы

## 8.6. Обзор литературы

Проблема численного решения систем линейных уравнений широко рассматривается в литературе. Для освоения тематики могут быть рекомендованы работы [2-4,23,72,85]. Широкое обсуждение вопросов параллельных вычислений для решения данного типа задач выполнено в работе [44,50].

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа [45]. В данной работе рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScALAPACK.

## **8.7. Контрольные вопросы**

1. Что представляет собой система линейных уравнений? Какие типы систем Вам известны? Какие методы могут быть использованы для решения систем разных типов?
2. В чем состоит постановка задачи решения системы линейных уравнений?
3. В чем идея параллельного обобщения метода Гаусса?
4. Какие показатели эффективности для параллельного варианта метода Гаусса?
6. В чем состоит схема программной реализации параллельного варианта метода Гаусса?
7. В чем состоит идея параллельной реализации метода сопряженных градиентов?
8. Какой из алгоритмов обладает лучшими показателями ускорения?

## **8.8. Задачи и упражнения**

1. Выполните анализ эффективности параллельных вычислений в отдельности для прямого и обратного этапов метода Гаусса. Оцените, на каком этапе происходит большее снижение показателей.
2. Выполните разработку параллельного варианта метода Гаусса при вертикальном разбиении матрицы по столбцам. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.
3. Выполните разработку параллельных вариантов методов Якоби и Зейделя решения систем линейных уравнений (см. например, [2-3,72]). Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.

Глава 9. Сортировка данных.....	1
9.1. Основы сортировки и принципы распараллеливания.....	2
9.2. Пузырьковая сортировка .....	3
9.2.1. Последовательный алгоритм пузырьковой сортировки.....	3
9.2.2. Метод чет-нечетной перестановки.....	7
9.2.3. Базовый параллельный алгоритм пузырьковой сортировки .....	7
9.2.4. Блочный параллельный алгоритм пузырьковой сортировки .....	13
9.3. Сортировка Шелла .....	20
9.3.1. Последовательный алгоритм.....	20
9.3.2. Организация параллельных вычислений.....	21
9.3.3. Анализ эффективности.....	22
9.3.4. Программная реализация .....	22
9.3.5. Результаты вычислительных экспериментов.....	26
9.4. Быстрая сортировка.....	29
9.4.1. Последовательный алгоритм быстрой сортировки .....	29
9.4.2. Параллельный алгоритм быстрой сортировки.....	31
9.4.3. Обобщенный алгоритм быстрой сортировки.....	39
9.4.4. Сортировка с использованием регулярного набора образцов .....	44
9.5. Краткий обзор главы .....	52
9.6. Обзор литературы .....	54
9.7. Контрольные вопросы .....	54
9.8. Задачи и упражнения.....	54

## Глава 9. Сортировка данных

*Сортировка* является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, \dots, a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a'_1, a'_2, \dots, a'_n) : a'_1 \leq a'_2 \leq \dots \leq a'_n\}$$

(здесь и далее все пояснения для краткости будут даваться только на примере упорядочивания данных по возрастанию).

Возможные способы решения этой задачи широко обсуждаются в литературе; один из наиболее полных обзоров *алгоритмов сортировки* содержится в работе [71], может быть рекомендована также работа [17].

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T \sim n^2.$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$$T \sim n \log_2 n.$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из  $n$  значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких ( $p > 1$ ) вычислительных элементов (процессоров или ядер). Исходный упорядочиваемый набор в этом случае «разделяется» на блоки, которые могут обрабатываться вычислительными элементами параллельно.

Оставляя подробный анализ проблемы сортировки для отдельного рассмотрения, здесь основное внимание мы уделим изучению параллельных способов выполнения для ряда широко известных методов *внутренней сортировки*, когда все упорядочиваемые данные могут быть размещены полностью в оперативной памяти ЭВМ.

## 9.1. Основы сортировки и принципы распараллеливания

При внимательном рассмотрении способов упорядочивания данных, применяемых в алгоритмах сортировки, можно обратить внимание, что многие методы основаны на применении одной и той же *базовой операции "Сравнить и переставить"* (*compare-exchange*), состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановки этих значений, если их порядок не соответствует условиям сортировки.

```
// базовая операция сортировки
if ( A[i] > A[j] ) {
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

Пример 9.1.      Операция "Сравнить и переставить"

Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения собственно и проявляется различие алгоритмов сортировки.

Рассмотренная выше базовая операция сортировки может быть надлежащим образом обобщена для случая, когда упорядочиваемые данные разделены на  $p$ ,  $p > 1$ , частей (блоков). Выделяемые при этом блоки имеют, как правило, одинаковый размер, и содержат в этом случае  $n/p$  элементов.

Блоки обычно упорядочиваются в самом начале сортировки - как можно заметить, блоки могут упорядочиваться независимо друг от друга, т.е. параллельно. Далее, следуя схеме одноэлементного сравнения, упорядочение содержимого блоков  $A_i$  и  $A_{i+1}$  может быть осуществлено следующим образом:

- объединить блоки  $A_i$  и  $A_{i+1}$  в один отсортированный блок двойного размера (при исходной упорядоченности блоков  $A_i$  и  $A_{i+1}$  процедура их объединения сводится к быстрой операции слияния упорядоченных наборов данных),
- разделить полученный двойной блок на две равные части:

$$[A_i \cup A_{i+1}]_{copm} = A'_i \cup A'_{i+1} : \forall a'_i \in A'_i, \forall a'_j \in A'_{i+1} \Rightarrow a'_i \leq a'_j.$$

Рассмотренная процедура обычно именуется в литературе как *операция "Сравнить и разделить"* (*compare-split*). Следует отметить, что сформированные в результате такой процедуры блоки совпадают по размеру с исходными блоками  $A_i$  и  $A_{i+1}$ , являются упорядоченными и все значения, расположенные в блоке  $A'_i$ , не превышают значений в блоке  $A'_{i+1}$ .

Трудоемкость рассмотренной операции при использовании быстрых алгоритмов сортировки является равной:

$$T' = 2(n/p) \log_2(n/p) + 2(n/p),$$

в то время как обычное упорядочивание данных, располагаемых в двух блоках, требует выполнения

$$T'' = (2n/p) \log_2(2n/p)$$

операций. Приведенные оценки показывают, что вычислительная сложность операции "Сравнить и разделить" при прочих равных условий является меньшей. Кроме того, "блочность" данной операции может привести к более эффективному использованию кэш-памяти процессоров, что позволит еще больше повысить эффективность выполнения алгоритмов сортировки.

Определенная выше операция "Сравнить и разделить" может быть использована в качестве *базовой подзадачи* для организации параллельных вычислений. Как следует из построения, количество таких подзадач параметрически зависит от числа имеющихся блоков и, таким образом, проблема масштабирования вычислений для параллельных алгоритмов сортировки практически отсутствует. Вместе с тем следует отметить, что относящиеся к подзадачам блоки данных изменяются в ходе выполнения сортировки. В простых случаях размер блоков данных в подзадачах остается неизмененным. В более сложных ситуациях (как, например, в алгоритме быстрой сортировки – см. 9.4) объем блоков может различаться, что может приводить к нарушению равномерной вычислительной загрузки вычислительных элементов.

## 9.2. Пузырьковая сортировка

*Алгоритм пузырьковой сортировки* (см., например, [17,71]) является одним из наиболее широко известных методов упорядочивания данных, однако в силу своей низкой эффективности в основном используется только в учебных целях.

### 9.2.1. Последовательный алгоритм пузырьковой сортировки

**1. Общая схема метода.** Последовательный алгоритм пузырьковой сортировки сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности

$$(a_1, a_2, \dots, a_n)$$

алгоритм сначала выполняет  $n-1$  базовых операций "сравнения-обмена" для последовательных пар элементов

$$(a_1, a_2), (a_2, a_3), \dots, (a_{n-1}, a_n).$$

В результате после первой итерации алгоритма самый большой элемент перемещается ("всплывает") в конец последовательности. Далее последний элемент в преобразованной последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности

$$(a'_1, a'_2, \dots, a'_{n-1}).$$

Как можно увидеть, последовательность будет отсортирована после  $n-1$  итерации. Эффективность пузырьковой сортировки может быть улучшена, если завершать алгоритм в случае отсутствия каких-либо изменений сортируемой последовательности данных в ходе какой-либо итерации сортировки.

```
// Алгоритм 9.1.  
// Serial bubble sort algorithm  
BubbleSort(double A[], int n) {  
    for (i=0; i<n-1; i++)  
        for (j=0; j<n-1-i; j++)  
            compare_exchange(A[j], A[j+1]);  
}
```

Алгоритм 9.1. Последовательный алгоритм пузырьковой сортировки

**2. Анализ эффективности.** При анализе эффективности последовательного алгоритма пузырьковой сортировки снова используем подход, примененный в п. 6.5.4.

Итак, время выполнения алгоритма складывается из времени, которое тратится непосредственно на вычисления, и времени, необходимого на чтение данных из оперативной памяти в кэш процессора. Оценим количество вычислительных операций, выполняемых в ходе пузырьковой сортировки. На первой итерации выполняется  $n-1$  операций сравнения-обмена, на второй итерации –  $n-2$  операции и т.д. Для сортировки всего набора данных необходимо выполнить  $n-1$  итерацию. Следовательно, общее время выполнения вычислений составляет:

$$T_1(\text{calc}) = \sum_{i=1}^{n-1} (n-i) \cdot t = \frac{n^2 - n}{2} \cdot t, \quad (9.1)$$

где  $t$  есть время выполнения базовой операции сортировки.

Если размер сортируемого набора данных настолько велик, что не может полностью поместиться в кэш, то каждый проход по массиву вызывает повторное считывание значений. Действительно, при движении по массиву и сравнении пар значений со всеми возрастающими индексами необходимо снова и снова считывать необходимые данные из оперативной памяти в кэш. Поскольку размер кэша ограничен, то считывание новых данных приведет к вытеснению данных, прочитанных ранее. Общее количество читаемых данных из памяти совпадает с числом операций сравнения-обмена. Таким образом, время, которое требуется на чтение необходимых данных в кэш, может быть ограничено сверху величиной:

$$T_1(\text{mem}) = \frac{n^2 - n}{2} \cdot 64 / b \quad (9.2)$$

где  $\beta$  – эффективная скорость доступа к оперативной памяти (коэффициент 64 введен для учета факта, что считывание данных из памяти осуществляется кэш-строками размером в 64 байт).

Если, как и в предыдущих разделах, учесть латентность памяти, то время доступа к памяти можно получить по следующей формуле:

$$T_1(\text{mem}) = \frac{n^2 - n}{2} \cdot (a + 64/b) \quad (9.3)$$

С учетом полученных соотношений, общее время выполнения последовательного алгоритма пузырьковой сортировки в худшем случае может быть вычислено по формуле:

$$T_1 = \frac{n^2 - n}{2} \cdot t + \frac{n^2 - n}{2} \cdot (a + 64/b) \quad (9.4)$$

Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см п. 6.5.4):

$$T_1 = \frac{n^2 - n}{2} \cdot t + g \cdot \frac{n^2 - n}{2} \cdot (a + 64/b) \quad (9.5)$$

**3. Программная реализация.** Представим возможный вариант последовательной программы, выполняющей алгоритм пузырьковой сортировки.

**1. Главная функция программы.** Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 9.1.
// Serial bubble sort algorithm
void main(int argc, char* argv[]) {
    double *pData;      // Data to be sorted
    int Size;           // Size of data to be sorted

    // Data initialization
    ProcessInitialization(pData, Size);
```

```

// Serial bubble sort
SerialBubbleSort(pData, Size);

// Program termination
ProcessTermination(pData);
}

```

**2. Функция ProcessInitialization.** Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества сортируемых данных, выделения памяти для сортируемых данных и для заполнения этой памяти начальными, неупорядоченными, значениями. Начальные значения задаются в функции *RandomDataInitialization*.

```

// Function for memory allocation and data initialization
void ProcessInitialization(double* &pData, int &Size) {
    do {
        printf ("Enter the array size:\n");
        scanf ("%d", &Size);
    } while (Size<=1);
    printf ("Chosen array size = %d\n", Size);

    pData = new double [Size];
    RandomDataInitialization(pData, Size);
}

```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

**3. Функция SerialBubbleSort.** Данная функция выполняет последовательный алгоритм пузырьковой сортировки.

```

// Function for serial bubble sorting
void SerialBubbleSort(double* pData, int Size) {
    double temp;
    for (int i=0; i<Size-1; i++)
        for (int j=1; j<Size-i; j++)
            if (pData[j-1]>pData[j]) {
                temp = pData[j];
                pData[j] = pData[j-1];
                pData[j-1] = temp;
            }
}

```

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

**4. Результаты вычислительных экспериментов.** Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Для снижения сложности построения теоретических оценок времени выполнения алгоритмов при компиляции и построении программ для проведения вычислительных экспериментов функция оптимизации кода компилятором была отключена (результаты оценки влияния компиляторной оптимизации на эффективность программного кода приведены в п. 7.2.4).

Оценка времени одной операции сравнения-обмена  $\tau$  осуществляется при помощи выполнения последовательного алгоритма пузырьковой сортировки при малых объемах данных, таких, чтобы весь массив, подлежащий сортировке, полностью помещается в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений массив заполняется

случайными числами. Выполнение этого действия гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, можно определить время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение  $\tau$ , равное 19,975 нс.

Оценки времени латентности  $\alpha$  и величины пропускной способности канала доступа к оперативной памяти  $\beta$  проводилась в п. 6.5.4 и определены для используемого вычислительного узла как  $\alpha = 8,31$  нс и  $\beta = 12,44$  Гб/с.

В таблице 9.1 и на рис. 9.1 представлены результаты сравнения времени выполнения  $T_1$  последовательного алгоритма пузырьковой сортировки со временем  $T_1^*$ , полученным при помощи модели (9.5). Частота кэш промахов, измеренная с помощью системы VPS, для одного потока значение этой величины было оценено как 0,0037.

**Таблица 9.1.** Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма пузырьковой сортировки

Размер массива	$T_1$	$T_1^*$ (calc) (модель)	Модель 9.4 – оценка сверху		Модель 9.5 – уточненная оценка	
			$T_1^*$ (mem)	$T_1^*$	$T_1^*$ (mem)	$T_1^*$
10000	1,0209	0,9987	0,6550	1,6537	0,0024	1,0011
20000	4,0854	3,9948	2,6201	6,6149	0,0097	4,0045
30000	9,1920	8,9885	5,8954	14,8839	0,0218	9,0103
40000	16,3296	15,9796	10,4808	26,4604	0,0388	16,0184
50000	25,5062	24,9683	16,3764	41,3446	0,0606	25,0288
60000	36,7364	35,9544	23,5821	59,5365	0,0873	36,0417
70000	50,0023	48,9381	32,0979	81,0359	0,1188	49,0568
80000	65,3137	63,9192	41,9239	105,8431	0,1551	64,0743
90000	82,6420	80,8979	53,0600	133,9578	0,1963	81,0942
100000	102,0637	99,8740	65,5062	165,3802	0,2424	100,1164

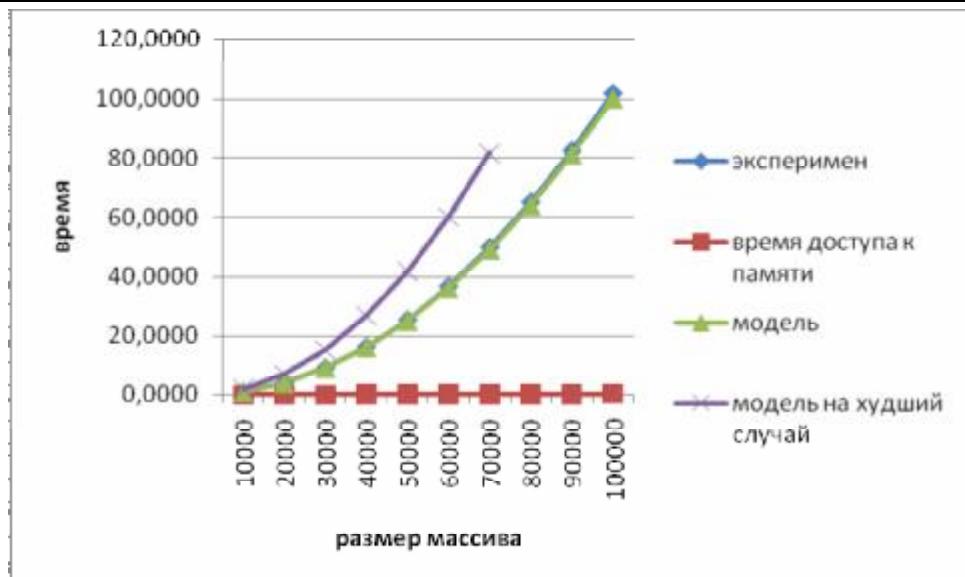


Рис. 9.1. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных

### 9.2.2. Метод чет-нечетной перестановки

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения обычно используется не сам этот алгоритм, а его модификация, известная в литературе как метод *чет-нечетной перестановки* (*odd-even transposition*) – см., например, [72]. Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары

$$(a_1, a_2), (a_3, a_3), \dots, (a_{n-1}, a_n) \text{ (при четном } n\text{),}$$

а на четных итерациях обрабатываются элементы

$$(a_2, a_3), (a_4, a_5), \dots, (a_{n-2}, a_{n-1}).$$

После  $n$ -кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

```
// Function for serial odd-even transposition
void OddEvenSort ( double* pData, int Size ) {
    double temp;
    int upper_bound;
    if (Size%2==0)
        upper_bound = Size/2-1;
    else
        upper_bound = Size/2;

    for (int i=0; i<Size; i++) {
        if (i%2 == 0) // even iteration
            for (int j=0; j<Size/2; j++)
                compare_exchange(pData[2*j], pData[2*j+1]);
        else          // odd iteration
            for (int j=0; j<upper_bound; j++)
                compare_exchange(pData[2*j+1], pData[2*j+2]);
    }
}
```

### 9.2.3. Базовый параллельный алгоритм пузырьковой сортировки

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – несмотря на то, что четные и нечетные итерации должны выполняться строго последовательно, сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. Поскольку все вычислительные элементы имеют прямой доступ к каждому значению в сортируемом массиве, сравнение значений  $a_i$  и  $a_j$  может быть выполнено любым вычислительным элементом. При наличии нескольких вычислительных элементов появляется возможность одновременно выполнять операцию «Сравнить и переставить» над несколькими парами значений.

**1. Анализ эффективности.** Как и ранее, при анализе эффективности базового параллельного алгоритма пузырьковой сортировки будем предполагать, что время выполнения складывается из времени вычислений (которые могут быть выполнены вычислительными элементами параллельно) и времени, необходимого на загрузку

необходимых данных из оперативной памяти в кэш. Доступ к памяти осуществляется строго последовательно.

Для сортировки массива данных методом чет-нечетной перестановки требуется выполнение  $n$  итераций алгоритма, на каждой из которых параллельно выполняется сравнение  $n/2$  пар значений. Значит, время выполнения вычислений составляет:

$$T_p(\text{calc}) = \frac{n^2 - n}{2p} \cdot t \quad (9.6)$$

Если объем сортируемых данных настолько велик, что не может быть полностью помещен в кэш вычислительного элемента, то на каждой итерации методы выполняются постепенное вытеснение значений, располагаемых в начале массива для того, чтобы записать на их место значения, располагаемые в конце массива. Таким образом, для перехода к выполнению следующей итерации необходимо будет снова считывать значения из начала сортируемого набора. Таким образом, на каждой итерации происходит повторное считывание всего массива данных из оперативной памяти в кэш, и затраты на доступ к памяти составляют:

$$T_p(\text{mem}) = \frac{n^2 - n}{2} \cdot \frac{64}{b} \quad (9.7)$$

Если, как и в предыдущих разделах, учесть латентность памяти, то время доступа к памяти можно получить по следующей формуле:

$$T_p(\text{mem}) = \frac{n^2 - n}{2} \cdot (a + 64/b) \quad (9.8)$$

При получении итоговой оценки времени выполнения базового параллельного алгоритма пузырьковой сортировки необходимо также учитывать затраты на организацию и закрытие параллельных секций:

$$T_p = \frac{n^2 - n}{2p} \cdot t + \frac{n^2 - n}{2} \cdot (a + 64/b) + nd, \quad (9.9)$$

где  $\delta$  есть накладные расходы на организацию параллельности на каждой итерации алгоритма.

Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см п. 6.5.4):

$$T_p = \frac{n^2 - n}{2p} \cdot t + g \cdot \frac{n^2 - n}{2} \cdot (a + 64/b) + nd, \quad (9.10)$$

**2. Программная реализация.** Рассмотрим возможный вариант реализации параллельного варианта метода пузырьковой сортировки. Используя OpenMP, получение параллельного алгоритма из последовательного достигается путем добавления двух директив препроцессора, при помощи которых распараллеливаются циклы сравнения пар значений на четных и нечетных итерациях метода чет-нечетной перестановки.

```
// Function for parallel odd-even transposition
void ParallelOddEvenSort ( double* pData, int Size ) {
    double temp;
    int upper_bound;
    if (Size%2==0)
        upper_bound = Size/2-1;
    else
        upper_bound = Size/2;

    for (int i=0; i<Size; i++) {
        if (i%2 == 0) // Even iteration
#pragma omp parallel for
            for (int j=0; j<Size/2; j++)
                compare_exchange(pData[2*j], pData[2*j+1]);
        else          // Odd iteration
    }
}
```

```

#pragma omp parallel for
    for (int j=0; j<upper_bound; j++)
        compare_exchange(pData[2*j+1], pData[2*j+2]);
}

```

**3. Результаты вычислительных экспериментов.** Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 9.2. Времена выполнения алгоритмов указаны в секундах.

**Таблица 9.2.** Результаты вычислительных экспериментов для параллельного метода пузырьковой сортировки (при использовании двух и четырех вычислительных ядер)

Размер массива	Последовательный алгоритм		Параллельный алгоритм					
	Пузырьковая сортировка ( $T_I$ )	Быстрая сортировка ( $T_I'$ )	$T_2$	$T_4$	$T_I/T_2$	$T_I'/T_2$	$T_I/T_4$	$T_I'/T_4$
10000	0,9287	0,0029	0,6361	0,3463	1,4599	0,0045	2,6819	0,0083
20000	3,7084	0,0062	2,4428	1,3437	1,5181	0,0025	2,7598	0,0046
30000	8,3574	0,0100	5,4297	2,9288	1,5392	0,0018	2,8536	0,0034
40000	14,8543	0,0133	9,5790	5,1394	1,5507	0,0014	2,8903	0,0026
50000	23,2230	0,0172	14,911	7,9255	1,5573	0,0012	2,9302	0,0022
60000	33,4188	0,0209	21,428	11,369	1,5596	0,0010	2,9393	0,0018
70000	45,6135	0,0245	29,103	15,389	1,5673	0,0008	2,9639	0,0016
80000	59,5642	0,0278	37,956	20,034	1,5693	0,0007	2,9731	0,0014
90000	75,3640	0,0318	47,982	25,314	1,5707	0,0007	2,9771	0,0013
100000	93,0108	0,0366	59,183	31,180	1,5716	0,0006	2,9829	0,0012

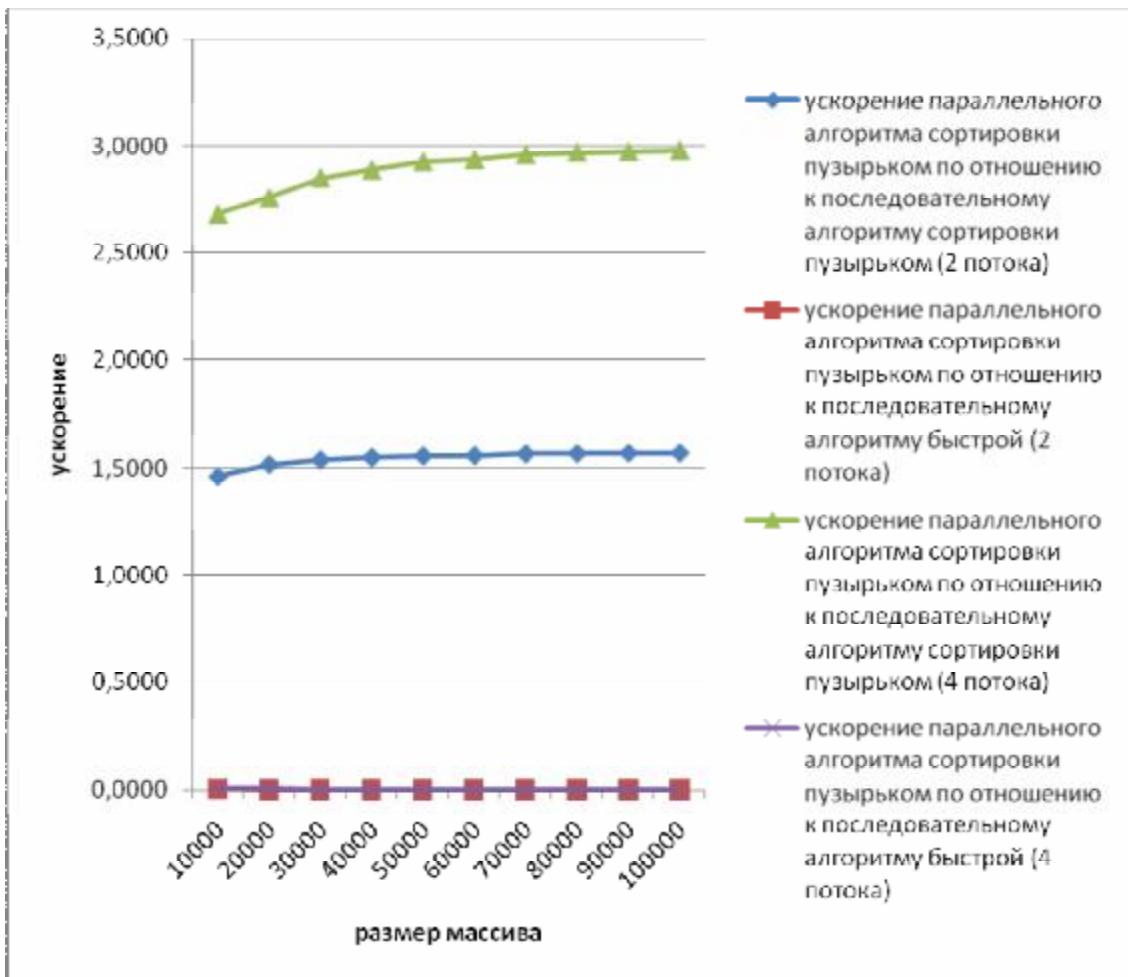


Рис. 9.2. Зависимость ускорения от количества исходных данных при выполнении параллельного метода пузырьковой сортировки

При проведении анализа эффективности алгоритм пузырьковой сортировки позволяет продемонстрировать следующий важный момент. Как уже отмечалось в начале данного раздела, использованный для распараллеливания последовательный метод упорядочивания данных характеризуется квадратичной зависимостью сложности от числа упорядочиваемых данных, т.е.  $T_1 \sim n^2$ . Однако применение подобной оценки сложности последовательного алгоритма приведет к искажению исходного целевого назначения критерии качества параллельных вычислений – показатели эффективности в этом случае будут характеризовать используемый способ параллельного выполнения данного конкретного метода сортировки, а не результативность использования параллелизма для задачи упорядочивания данных в целом как таковой. Различие состоит в том, что для сортировки могут быть применены более эффективные последовательные алгоритмы, трудоемкость которых имеет порядок

$$T_1 = n \log_2 n ,$$

и для сравнения, насколько быстрее могут быть упорядочены данные при использовании параллельных вычислений, в обязательном порядке должна использоваться именно данная оценка сложности. Как основной результат выполненных рассуждений, можно сформулировать утверждение о том, что *при определении показателей ускорения и эффективности параллельных вычислений в качестве оценки сложности последовательного способа решения рассматриваемой задачи следует использовать трудоемкость наилучших последовательных алгоритмов*. Параллельные методы решения задач должны сравниваться с наиболее быстродействующими последовательными способами вычислений!

Как следует из данных, приведенных в таблице 9.2 и на рис. 9.2 ускорение полученного параллельного метода по отношению к наиболее эффективному последовательному совершенно неудовлетворительно. Поэтому при разработке параллельных методов сортировки следует использовать за основу более эффективные методы упорядочивания данных.

В таблицах 9.3 и 9.4 и на рис. 9.3 и 9.4 представлены результаты сравнения времени выполнения  $T_p$  базового параллельного метода пузырьковой сортировки с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (9.10). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,197, а для четырех потоков значение этой величины было оценено как 0,101.

Как отмечалось в главе 6, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах сортируемого массива составляет порядка 6%.

**Таблица 9.3.** Сравнение экспериментального и теоретического времени выполнения параллельного метода пузырьковой сортировки с использованием двух потоков

Размер массива	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 9.9 – оценка сверху		Модель 9.10 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
10000	0,6361	0,5018	0,6550	1,1568	0,1290	0,6309
20000	2,4428	2,0024	2,6201	4,6225	0,5162	2,5186
30000	5,4297	4,5017	5,8954	10,3971	1,1614	5,6631
40000	9,5790	7,9998	10,4808	18,4806	2,0647	10,0645
50000	14,9119	12,4966	16,3764	28,8730	3,2261	15,7228
60000	21,4280	17,9922	23,5821	41,5743	4,6457	22,6379
70000	29,1039	24,4865	32,0979	56,5844	6,3233	30,8098
80000	37,9563	31,9796	41,9239	73,9035	8,2590	40,2386
90000	47,9824	40,4714	53,0600	93,5314	10,4528	50,9242
100000	59,1836	49,9620	65,5062	115,4682	12,9047	62,8667



Рис. 9.3. График зависимости экспериментального и теоретического времени выполнения базового параллельного метода пузырьковой сортировки от объема исходных данных при использовании двух потоков

**Таблица 9.4.** Сравнение экспериментального и теоретического времени выполнения параллельного метода пузырьковой сортировки с использованием четырех потоков

Размер массива	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 9.9 – оценка сверху		Модель 9.10 – уточненная оценка	
			$T_p^* \text{ (tem)}$	$T_p^*$	$T_p^* \text{ (tem)}$	$T_p^*$
10000	0,3463	0,2522	0,6550	0,9072	0,0662	0,3183
20000	1,3437	1,0037	2,6201	3,6238	0,2646	1,2683
30000	2,9288	2,2546	5,8954	8,1500	0,5954	2,8501
40000	5,1394	4,0049	10,4808	14,4857	1,0586	5,0635
50000	7,9255	6,2546	16,3764	22,6309	1,6540	7,9086
60000	11,3697	9,0036	23,5821	32,5857	2,3818	11,3854
70000	15,3899	12,2520	32,0979	44,3499	3,2419	15,4939
80000	20,0343	15,9998	41,9239	57,9237	4,2343	20,2341
90000	25,3142	20,2470	53,0600	73,3069	5,3591	25,6060
100000	31,1809	24,9935	65,5062	90,4997	6,6161	31,6096

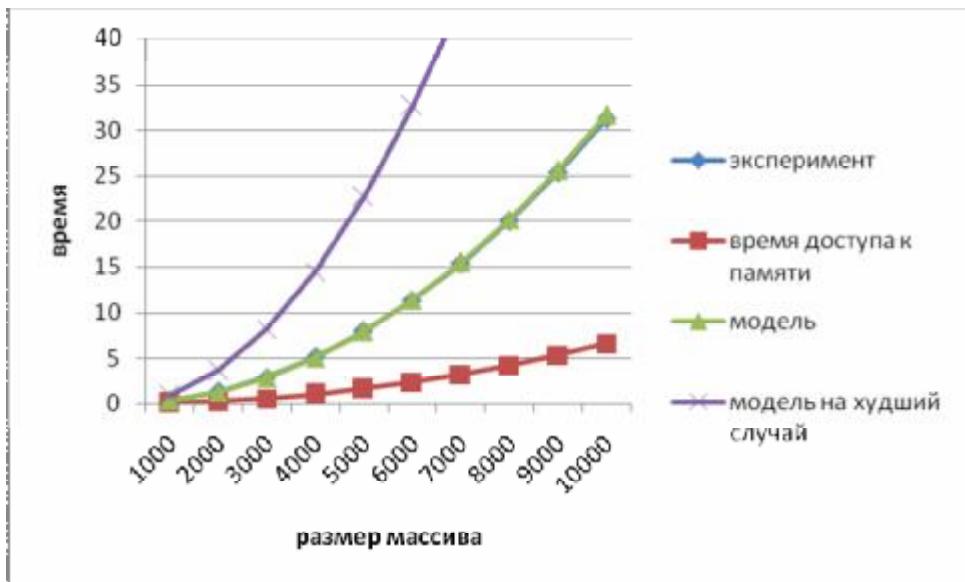


Рис. 9.4. График зависимости экспериментального и теоретического времени выполнения базового параллельного метода пузырьковой сортировки от объема исходных данных при использовании четырех потоков

#### 9.2.4. Блочный параллельный алгоритм пузырьковой сортировки

**1. Принципы распараллеливания.** Рассмотрим ситуацию, когда количество вычислительных элементов является меньшим числа упорядочиваемых значений ( $p < n$ ). Разделим сортируемый массив на блоки данных размера  $n/p$ . На первом этапе параллельного метода пузырьковой сортировки каждый вычислительный элемент выполняет сортировку одного из блоков данных при помощи какого-либо быстрого алгоритма (например, при помощи алгоритма быстрой сортировки – см. раздел 9.4.1); эти действия могут быть выполнены параллельно. На следующем этапе выполняется параллельный алгоритм чет-нечетной перестановки над блоками данных:

- На четных итерациях алгоритма вычислительные элементы с четными индексами  $2i$  выполняют операцию "Сравнить и разделить" для двух упорядоченных блоков с номерами  $2i$  и  $2i+1$ ,
- На нечетных итерациях алгоритма вычислительные элементы с нечетными индексами  $2i+1$  выполняют операцию "Сравнить и разделить" для двух упорядоченных блоков с номерами  $2i+1$  и  $2i+2$ .

После выполнения  $p$  итераций чет-нечетной перестановки исходный массив оказывается упорядоченным.

Для пояснения такого параллельного способа сортировки в табл. 9.5 приведен пример упорядочения данных при  $n=16$ ,  $p=4$  (т.е. блок значений на каждом вычислительном элементе содержит  $n/p=4$  элемента). В первом столбце таблицы приводится номер и тип итерации метода, перечисляются пары номеров блоков данных, для которых параллельно выполняются операции слияния. Взаимодействующие пары блоков выделены в таблице двойной рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

**Таблица 9.5.** Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итерации	Вычислительные элементы			
	0	1	2	3
1				

Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
0 чет (1,2),(3,4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
1 нечет (2,3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
2 чет (1,2),(3,4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
3 нечет (2,3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

Заметим, однако, что при выполнении каждой итерации чет-нечетной перестановки блоков, задействованными оказываются только  $p/2$  вычислительных элементов, в то время как остальные вычислительные элементы пристаивают. Это приводит к снижению общей эффективности параллельного алгоритма. Для повышения эффективности параллельного метода пузырьковой сортировки разделим исходный массив не на  $p$ , а на  $2p$  блоков, каждый из которых содержит  $n/2p$  значений. На первом этапе сортировки каждый вычислительный элемент выполняет сортировку двух последовательных блоков данных при помощи какого-либо быстрого алгоритма. На следующем этапе выполняется параллельный алгоритм чет-нечетной перестановки над блоками данных меньшего размера:

- На четных итерациях алгоритма каждый вычислительный элемент с индексом  $i$  выполняет операцию "Сравнить и разделить" для двух упорядоченных блоков с номерами  $2i$  и  $2i+1$ ,
- На нечетных итерациях алгоритма каждый вычислительный элемент с индексом  $i$  выполняет операцию "Сравнить и разделить" двух упорядоченных блоков с номерами  $2i+1$  и  $2i+2$ .

После выполнения  $2p$  итераций чет-нечетной перестановки исходный массив оказывается упорядоченным.

В общем случае выполнение параллельного метода может быть прекращено, если после выполнения очередной итерации массив оказался отсортированным. Как результат, общее количество итераций может быть сокращено, и для фиксации таких моментов необходимо, чтобы ведущий поток (*master thread*) определял состояние набора данных после выполнения каждой итерации сортировки.

**2. Анализ эффективности.** При построении оценок эффективности блочного параллельного алгоритма пузырьковой сортировки будут использованы соотношения, описывающие трудоемкость последовательного алгоритма быстрой сортировки (см. п. 9.4.1). Это объясняется тем, что для первоначальной сортировки блоков данных каждым вычислительным элементом использовался именно алгоритм быстрой сортировки.

Определим теперь сложность рассмотренного параллельного алгоритма упорядочивания данных. Как отмечалось ранее, на начальной стадии работы метода каждый вычислительный элемент проводит упорядочивание двух блоков данных (размер каждого блока при равномерном распределении данных является равным  $n/2p$ ). Пусть данная начальная сортировка выполнена при помощи быстродействующих алгоритмов упорядочивания данных, тогда трудоемкость начальной стадии вычислений можно определить выражением вида (см. оценку (9.20)):

$$T_p^1 = 2 \cdot 1.4 \cdot (n/2p) \log_2(n/2p). \quad (9.11)$$

Далее на каждой выполняемой итерации параллельной сортировки каждый вычислительный элемент осуществляет объединение пары блоков при помощи процедуры слияния и затем разделения объединенного блока на две равные по размеру части. Общее количество итераций не превышает величины  $2p$ , и, как результат, общее количество операций этой части параллельных вычислений оказывается равным

$$T_p^2 = (2p) \cdot \left( 2 \frac{n}{2p} \right) = 2n. \quad (9.12)$$

Итак, время выполнения вычислений может быть получено при помощи выражения:

$$T_p(\text{calc}) = (2 \cdot 1,4 \cdot (n/2p) \log_2(n/2p) + 2n) \cdot t \quad (9.13)$$

где  $t$  есть время выполнения базовой операции сортировки.

Теперь оценим время, необходимое на чтение данных из оперативной памяти в кэш. При выполнении первого этапа алгоритма, который состоит в «локальной» сортировке блоков, при выполнении сортировки каждого блока вычислительный элемент считывает из оперативной памяти количество значений, определяемое оценкой (9.21) с поправкой на размер блока, который обрабатывается вычислительным элементом. Поскольку доступ к памяти осуществляется строго последовательно, то время, которое необходимо одному вычислительному элементу на считывание данных, нужно умножить на количество вычислительных элементов. Далее, на каждой итерации чет-нечетной перестановки блоков вычислительные элементы также считывают весь сортируемый набор данных снова. Таким образом, время, необходимое на загрузку необходимых данных из памяти составляет:

$$T_p(\text{mem}) = p \cdot \left( 2 \cdot 1,4 \cdot (n/2p) \log_2(n/2p) \frac{64}{b} \right) + 2p \cdot \frac{64n}{b} = (1,4 \log_2(n/2p) + 2p) \cdot \frac{64n}{b}. \quad (9.14)$$

Если, как и в предыдущих разделах, учесть латентность памяти, то время доступа к памяти можно подсчитать по следующей формуле:

$$T_p(\text{mem}) = (1,4 \log_2(n/2p) + 2p) \cdot n \cdot \left( a + \frac{64}{b} \right). \quad (9.15)$$

Кроме того, необходимо учитывать накладные расходы на организацию и закрытие параллельных секций: одна параллельная секция создается для выполнения локальной сортировки блоков, далее параллельные секции создаются для каждой итерации алгоритма чет-нечетной перестановки ( $2p$  итераций).

Итак, общее время выполнения параллельного алгоритма пузырьковой сортировки может быть вычислено в соответствии с выражением:

$$T_p = (2 \cdot 1,4 \cdot (n/2p) \log_2(n/2p) + 2n) \cdot t + (1,4 \log_2(n/2p) + 2p) \cdot n \cdot \left( a + \frac{64}{b} \right) + (2p+1) \cdot d \quad (9.16)$$

Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см п. 6.5.4):

$$T_p = (2 \cdot 1,4 \cdot (n/2p) \log_2(n/2p) + 2n) \cdot t + g(1,4 \log_2(n/2p) + 2p) \cdot n \cdot \left( a + \frac{64}{b} \right) + (2p+1) \cdot d \quad (9.17)$$

**3. Программная реализация.** Рассмотрим возможный вариант реализации блочного параллельного варианта метода пузырьковой сортировки.

Для упорядочения данных необходимо организовать синхронизированное выполнение отдельных этапов блочного параллельного метода пузырьковой сортировки между потоками параллельной программы. Так, нельзя приступить к выполнению операции слияния упорядоченных блоков до тех пор, пока все вычислительные элементы не выполнят локальную сортировку блоков. Наиболее простой способ организации синхронизации – выделить каждый этап в отдельную параллельную секцию при помощи директивы **parallel**. При закрытии параллельной секции автоматически выполняется синхронизация потоков. Таким образом, можно гарантировать, что к началу выполнения

очередного этапа алгоритма сортировки все потоки завершат выполнение предыдущего этапа.

В каждой параллельной секции необходимо иметь доступ к переменной, которая содержит число параллельных потоков, и к переменной-идентификатору текущего потока. Значение переменной *ThreadNum*, содержащей количество потоков, одинаково во всех потоках параллельной программы, используется потоками только для чтения, и, следовательно, эта переменная может быть общей для всех потоков. Переменная-идентификатор потока *ThreadID* имеет различное значение в различных потоках. Чтобы избежать многократного вызова функций библиотеки OpenMP для определения количества потоков и идентификаторов потоков объявим соответствующие переменные как глобальные. Переменную для хранения идентификатора потока определим как *threadprivate* – значение такой переменной, будучи однажды определено для каждого потока внутри параллельной секции, сохраняется во всех последующих параллельных секциях. Функция *InitializeParallelSections* служит для инициализации переменных *ThreadNum* и *ThreadID*.

```
int ThreadNum;      // Number of threads
int ThreadID;       // Thread identifier
#pragma omp threadprivate (ThreadID)

void InitializeParallelSections() {
#pragma omp parallel
{
    ThreadID = omp_get_thread_num();
#pragma omp single
    {
        ThreadNum = omp_get_num_threads();
    }
}
}
```

Функция *ParallelBubbleSort* выполняет блочный параллельный алгоритм пузырьковой сортировки. Дадим пояснения об использовании дополнительных структур данных. Массив *Index* хранит индексы первых элементов блоков данных. В массиве *BlockSize* хранятся размеры блоков данных. Таким образом, *i*-ый блок данных начинается с элемента *Index[i]* исходного массива и содержит *BlockSize[i]* элементов. Использование таких дополнительных массивов позволяет работать с блоками данных разного размера, например в случае, когда размер исходного массива некратен количеству вычислительных элементов.

```
// Function for parallel bubble sorting
void ParallelBubbleSort(double* pData, int Size) {
    InitializeParallelSections();
    int* Index = new int [ThreadNum*2];
    int* BlockSize = new int [ThreadNum*2];
    for (int i=0; i<2*ThreadNum; i++) {
        Index[i] = int((i*Size)/double(2*ThreadNum));
        if (i<2*ThreadNum-1)
            BlockSize[i] = int (((i+1)*Size)/double(2*ThreadNum)) - Index[i];
        else
            BlockSize[i] = Size-Index[i];
    }

    // Local sorting with quick algorithm
#pragma omp parallel
{
    LocalQuickSort(pData, Index[2*ThreadID],
                  Index[2*ThreadID]+BlockSize[2*ThreadID]-1);
    LocalQuickSort(pData, Index[2*ThreadID+1],
                  Index[2*ThreadID+1]+BlockSize[2*ThreadID+1]-1);
}
```

```

    }

    // Odd-even transposition of data blocks
    int Iter = 0;
    do {
#pragma omp parallel
    {
        if (Iter%2 == 0) { // Even iteration
            MergeBlocks(pData, Index[2*ThreadID], BlockSize[2*ThreadID],
                         Index[2*ThreadID+1], BlockSize[2*ThreadID+1]);
        }
        else { // Odd iteration
            if (ThreadID<ThreadNum-1)
                MergeBlocks(pData, Index[2*ThreadID+1], BlockSize[2*ThreadID+1],
                            Index[2*ThreadID+2], BlockSize[2*ThreadID+2]);
        }
   }// pragma omp parallel
    Iter++;
} while (!IsSorted(pData, Size));
}

```

Функция *MergeBlocks* выполняет слияние двух подряд идущих упорядоченных блоков заданного размера.

```

// Function for merging of two sorted blocks
void MergeBlocks(double* pData, int Index1, int BlockSize1, int Index2,
                 int BlockSize2) {
    double* pTempArray = new double [BlockSize1 + BlockSize2];
    int i1 = Index1, i2 = Index2, curr=0;
    while ((i1<Index1+BlockSize1) && (i2<Index2+BlockSize2)) {
        if (pData[i1] < pData[i2])
            pTempArray[curr++] = pData[i1++];
        else {
            pTempArray[curr++] = pData[i2++];
        }
    while (i1<Index1+BlockSize1)
        pTempArray[curr++] = pData[i1++];
    while (i2<Index2+BlockSize2)
        pTempArray[curr++] = pData[i2++];
    for (int i=0; i<BlockSize1+BlockSize2; i++)
        pData[Index1+i] = pTempArray[i];
    delete [] pTempArray;
}

```

Обратим внимание на следующий момент, связанный с реализацией метода. На каждой итерации параллельного алгоритма пузырьковой сортировки происходит слияние упорядоченных блоков данных, которое осуществляется при помощи функции *MergeBlocks*. Данная операция может быть выполнена более эффективно. Так, вместо определения нового массива *pTempArray* для выполнения слияния каждой пары блоков можно определить второй массив из *Size* элементов, в который будут записываться блоки данных при выполнении слияния. Тогда при переходе к следующей итерации алгоритма исходный и вновь определенный массив можно просто поменять ролями, избежав, тем самым, трудоемкой процедуры копирования данных. С другой стороны, такая реализация усложнило бы понимание программы и по этой причине в данном случае не используется.

Функция *IsSorted* проверяет, является ли массив отсортированным.

```

// Function for checking if the array is sorted
bool IsSorted(double* pData, int Size) {
    bool res = true;
    for (int i=1; (i<Size)&&(res); i++) {
        if (pData[i]<pData[i-1])
            res=false;
    }
}

```

```

    }
    return res;
}

```

**4. Результаты вычислительных экспериментов.** Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 9.6. Времена выполнения алгоритмов указаны в секундах.

**Таблица 9.6.** Результаты вычислительных экспериментов для блочного параллельного метода пузырьковой сортировки (при использовании двух и четырех вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
10000	0,0029	0,0018	1,6174	0,0013	2,2590
20000	0,0062	0,0037	1,6691	0,0025	2,4343
30000	0,0100	0,0060	1,6692	0,0038	2,6372
40000	0,0133	0,0079	1,6723	0,0053	2,5110
50000	0,0172	0,0107	1,6123	0,0069	2,5053
60000	0,0209	0,0123	1,6973	0,0081	2,5905
70000	0,0245	0,0143	1,7150	0,0094	2,5906
80000	0,0278	0,0165	1,6860	0,0108	2,5842
90000	0,0318	0,0190	1,6729	0,0122	2,6172
100000	0,0366	0,0208	1,7598	0,0142	2,5817

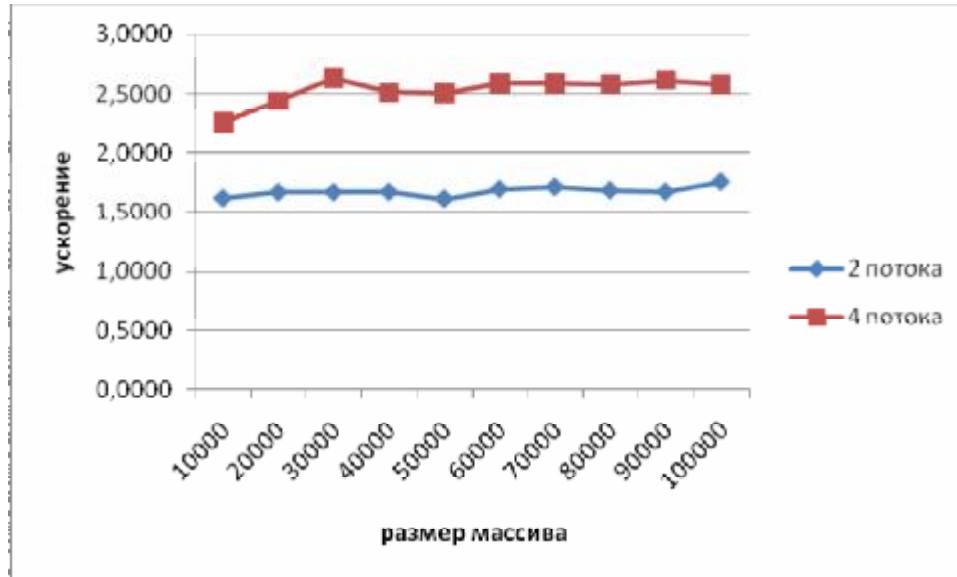


Рис. 9.5. Зависимость ускорения от количества исходных данных при выполнении параллельного метода пузырьковой сортировки

В таблицах 9.7 и 9.8 и на рис. 9.6 и 9.7 представлены результаты сравнения времени выполнения  $T_p$  параллельного метода пузырьковой сортировки с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (9.17). Частота кэш

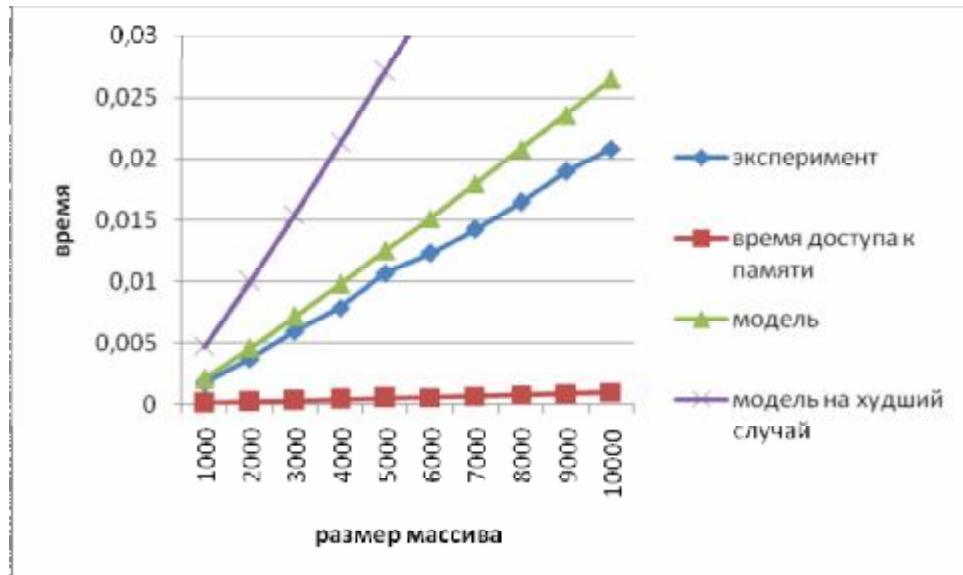
промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0322, а для четырех потоков значение этой величины было оценено как 0,0710.

Оценки времени латентности  $\alpha$  и величины пропускной способности канала доступа к оперативной памяти  $\beta$  проводилась в п. 6.5.4 и определены для используемого вычислительного узла как  $\alpha = 8,31$  нс и  $\beta = 12,4$  Гб/с.

Как и ранее, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

**Таблица 9.7.** Сравнение экспериментального и теоретического времени выполнения параллельного метода пузырьковой сортировки с использованием двух потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 9.16 – оценка сверху		Модель 9.17 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
10000	0,0018	0,0021	0,0026	0,0047	0,0001	0,0021
20000	0,0037	0,0044	0,0056	0,0100	0,0002	0,0046
30000	0,0060	0,0069	0,0087	0,0155	0,0003	0,0072
40000	0,0079	0,0094	0,0118	0,0213	0,0004	0,0098
50000	0,0107	0,0120	0,0151	0,0271	0,0005	0,0125
60000	0,0123	0,0146	0,0184	0,0330	0,0006	0,0152
70000	0,0143	0,0173	0,0218	0,0391	0,0007	0,0180
80000	0,0165	0,0200	0,0252	0,0451	0,0008	0,0208
90000	0,0190	0,0227	0,0286	0,0513	0,0009	0,0236
100000	0,0208	0,0255	0,0320	0,0575	0,0010	0,0265



**Рис. 9.6.** График зависимости экспериментального и теоретического времени выполнения блочного параллельного метода пузырьковой сортировки от объема исходных данных при использовании двух потоков

**Таблица 9.8.** Сравнение экспериментального и теоретического времени выполнения параллельного метода пузырьковой сортировки с использованием четырех потоков

Размер	$T_p$	$T_p^* (calc)$	Модель 9.16	Модель 9.17

матриц		(модель)	оценка сверху		уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
10000	0,0013	0,0011	0,0029	0,0041	0,0002	0,0013
20000	0,0025	0,0024	0,0062	0,0086	0,0004	0,0028
30000	0,0038	0,0037	0,0097	0,0134	0,0007	0,0044
40000	0,0053	0,0050	0,0132	0,0182	0,0009	0,0060
50000	0,0069	0,0064	0,0168	0,0232	0,0012	0,0076
60000	0,0081	0,0078	0,0205	0,0283	0,0015	0,0093
70000	0,0094	0,0092	0,0241	0,0334	0,0017	0,0109
80000	0,0108	0,0106	0,0279	0,0385	0,0020	0,0126
90000	0,0122	0,0121	0,0316	0,0437	0,0022	0,0143
100000	0,0142	0,0135	0,0354	0,0490	0,0025	0,0160



Рис. 9.7. График зависимости экспериментального и теоретического времени выполнения блочного параллельного метода пузырьковой сортировки от объема исходных данных при использовании четырех потоков

### 9.3. Сортировка Шелла

#### 9.3.1. Последовательный алгоритм

Общая идея сортировки Шелла (см., например, [17,71]) состоит в сравнении на начальных стадиях сортировки пар значений, располагаемых достаточно далеко друг от друга в упорядочиваемом наборе данных. Такая модификация метода сортировки позволяет быстро переставлять далекие неупорядоченные пары значений (сортировка таких пар обычно требует большого количества перестановок, если используется сравнение только соседних элементов).

Общая схема метода состоит в следующем. На первом шаге алгоритма происходит упорядочивание элементов  $n/2$  пар  $(a_i, a_{n/2+i})$  для  $1 \leq i \leq n/2$ . Далее на втором шаге упорядочиваются элементы в  $n/4$  группах из четырех элементов  $(a_i, a_{n/4+i}, a_{n/2+i}, a_{3n/4+i})$  для  $1 \leq i \leq n/4$ . На третьем шаге упорядочиваются элементы уже в  $n/4$  группах из восьми элементов и т.д. На последнем шаге упорядочиваются элементы сразу во всем массиве  $(a_1, a_2, \dots, a_n)$ . На каждом шаге для упорядочивания элементов в группах используется

метод сортировки вставками. Как можно заметить, общее количество итераций алгоритма Шелла является равным  $\log_2 n$ .

В более полном виде алгоритм Шелла может быть представлен следующим образом.

```
// Serial Shell sort algorithm
ShellSort(double A[], int n){
    int incr = n/2;
    while( incr > 0 ) {
        for ( int i=incr+1; i<n; i++ ) {
            j = i-incr;
            while ( j > 0 )
                if ( A[j] > A[j+incr] ){
                    swap(A[j], A[j+incr]);
                    j = j - incr;
                }
                else j = 0;
        }
        incr = incr/2;
    }
}
```

Алгоритм 9.2. Последовательный алгоритм сортировки Шелла

### 9.3.2. Организация параллельных вычислений

Для алгоритма Шелла может быть предложен параллельный аналог метода (см., например, [72]), если количество вычислительных элементов равно  $p=2^N$ . Разделим сортируемый набор данных на  $2p$  блоков равного размера. Таким образом, количество блоков данных является равным  $q=2^{N+1}$ . Образуем из блоков гиперкуб размерности  $N+1$ . Выполнение сортировки в этом случае может быть разделено на два последовательных этапа. На первом этапе ( $N+1$  итерация) осуществляется взаимодействие блоков данных, являющихся соседними в структуре гиперкуба (но эти блоки могут оказаться далекими при линейной нумерации; для установления соответствия двух систем нумерации процессоров может быть использован код Грея – см., например, [10]). На каждой итерации множество блоков разделяется на пары и для каждой пары блоков выполняется операция "Сравнить и разделить". Формирование пар блоков, взаимодействующих между собой, может быть обеспечено при помощи следующего простого правила – на каждой итерации  $i$ ,  $0 \leq i < N+1$ , парными становятся блоки, у которых различие в битовых представлении их номеров имеется только в позиции  $(N+1)-i$ . Отметим, что обработка различных пар взаимодействующих блоков может выполняться параллельно.

Второй этап состоит в реализации обычных итераций параллельного алгоритма чет-нечетной перестановки. Итерации данного этапа выполняются до прекращения фактического изменения сортируемого набора и, тем самым, общее количество  $L$  таких итераций может быть различным - от 2 до  $2p$ .

На рис. 9.8 показан пример сортировки массива из 16 элементов с помощью рассмотренного способа. Нужно заметить, что данные оказываются упорядоченными уже после первого этапа и нет необходимости выполнять чет-нечетную перестановку.

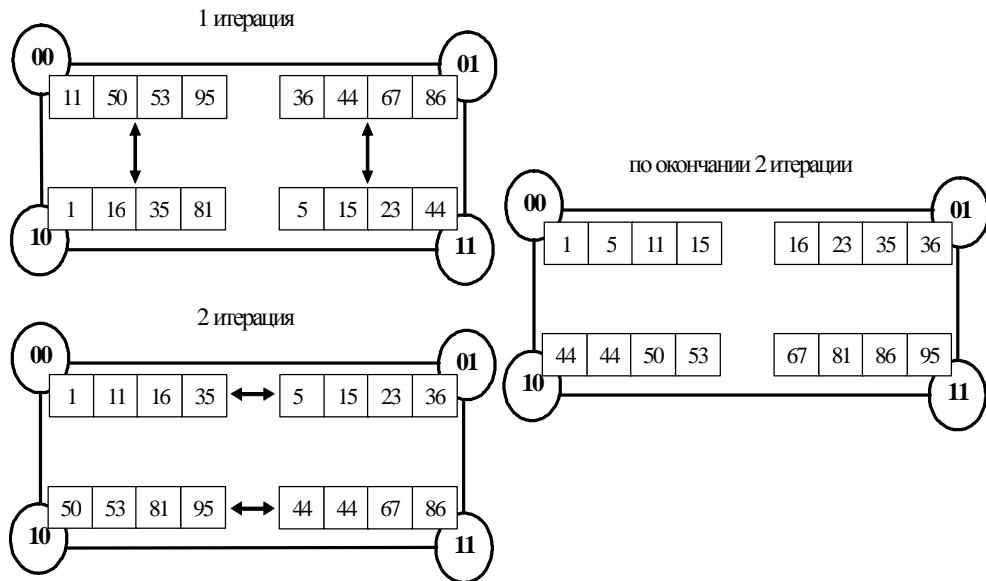


Рис. 9.8. Пример работы алгоритма Шелла для 2 вычислительных элементов (4 блока данных), номера блоков данных даны в битовом представлении

### 9.3.3. Анализ эффективности

Для оценки эффективности параллельного аналога алгоритма Шелла могут быть использованы соотношения, полученные для блочного параллельного метода пузырьковой сортировки (см. п. 9.2.4). При этом следует только учесть двухэтапность алгоритма Шелла – с учетом данной особенности общее время выполнения нового параллельного метода может быть определено при помощи выражения:

$$T_p = [2 \cdot 1.4 \cdot (n/2p) \log_2(n/2p) + (\log_2 p + L) \cdot n/p] \cdot t + [( \log_2 p + L ) + 1.4 \log_2(n/2p)] \cdot n \cdot (a + 64/b) + (\log_2 p + L + 1) \cdot d \quad (9.18)$$

Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$ :

$$T_p = [2 \cdot 1.4 \cdot (n/2p) \log_2(n/2p) + (\log_2 p + L) \cdot n/p] \cdot t + g[( \log_2 p + L ) + 1.4 \log_2(n/2p)] \cdot n \cdot (a + 64/b) + (\log_2 p + L + 1) \cdot d \quad (9.19)$$

Как можно заметить, эффективность параллельного варианта сортировки Шелла существенно зависит от значения  $L$  – при малом значении величины  $L$  новый параллельный способ сортировки выполняется быстрее, чем ранее рассмотренный алгоритм чет-нечетной перестановки.

### 9.3.4. Программная реализация

Рассмотрим возможный вариант реализации параллельного варианта метода сортировки Шелла. Как и при разработке программ, реализующих алгоритм пузырьковой сортировки, объявим несколько глобальных переменных: *ThreadNum* для определения количества потоков в параллельной программе, *DimSize* для определения размерности гиперкуба, который может быть составлен из заданного количества блоков данных, *ThreadID* для определения номера текущего потока. Создадим локальные копии переменной *ThreadID* при помощи директивы *threadprivate*.

Функция *InitializeParallelSections* определяет количество потоков *ThreadNum* и размерность виртуального гиперкуба *DimSize*, а также идентификатор потока *ThreadID*.

```
int ThreadNum; // Number of threads
int ThreadID; // Thread identifier
int DimSize; // Hypercube dimension

#pragma omp threadprivate(ThreadID)
```

```

void InitializeParallelSections() {
#pragma omp parallel
{
    ThreadID = omp_get_thread_num();
#pragma omp single
    ThreadNum = omp_get_num_threads();
}
DimSize = int(log10(double(ThreadNum))/log10(2.0));
}

```

Для установления соответствия между номером блока данных в линейной нумерации и его номером в структуре гиперкуба используется код Грэя (функция *GrayCode*). Для выполнения обратной операции (операции получения номера блока данных в линейной нумерации по его рангу в структуре гиперкуба) используется функция *ReverseGrayCode*:

```

// Function for calculation of the data block number in hypercube
int GrayCode (int RingID, int DimSize) {
    if ((RingID==0) && (DimSize==1))
        return 0;
    if ((RingID==1) && (DimSize==1))
        return 1;
    int res;
    if (RingID < (1<<(DimSize-1)))
        res = GrayCode(RingID, DimSize-1);
    else
        res = (1<<(DimSize-1))+GrayCode((1<<DimSize)-1-RingID, DimSize-1);
    return res;
}

// Function for calculation of the data block number in linear sequence
int ReverseGrayCode (int CubeID, int DimSize) {
    for (int i=0; i<(1<<DimSize); i++) {
        if (CubeID == GrayCode(i, DimSize))
            return i;
    }
}

```

Выполнение алгоритма может быть разделено на три этапа. На *первом этапе* вычислительные элементы параллельно выполняют сортировку блоков данных, при этом каждый вычислительный элемент сортирует 2 блока при помощи последовательного алгоритма быстрой сортировки. На *втором этапе* выполняются итерации алгоритма Шелла. И, наконец, на *третьем этапе* выполняются итерации метода чет-нечетной перестановки блоков до окончания фактического изменения сортируемого массива.

Процедура выполнения первого и третьего этапов является достаточно понятной, реализация же итераций алгоритма Шелла (*второй этап*) требует дополнительных пояснений. На каждой итерации алгоритма формируется массив пар номеров блоков в структуре гиперкуба, над которыми необходимо выполнить операцию «Сравнить и разделить». Число таких пар совпадает с числом вычислительных элементов. Пара с номером  $i$ ,  $0 \leq i < p$ , хранится в массиве *BlockPairs* в элементах с индексами  $2i$  и  $2i+1$ . Формирование массива пар осуществляется в функции *SetBlockPairs*:

```

// Function for block pairs determination
// "Compare-split" operation will be carried out for that pairs
void SetBlockPairs (int* BlockPairs, int Iter) {
    int PairNum = 0, FirstValue, SecondValue;
    bool Exist;
    for (int i=0; i<2*ThreadNum; i++) {
        FirstValue = GrayCode(i, DimSize);
        Exist = false;
        for (int j=0; (j<PairNum)&&(!Exist); j++)
            if (BlockPairs[2*j+1] == FirstValue)

```

```

        Exist = true;
if (!Exist) {
    SecondValue = FirstValue^(1<<(DimSize-Iter-1));
    BlockPairs[2*PairNum] = FirstValue;
    BlockPairs[2*PairNum+1] = SecondValue;
    PairNum++;
} // if
} // for
}

```

Далее для каждой пары блоков необходимо определить вычислительный элемент, который будет выполнять операцию «Сравнить и разделить». Возможный способ определения номера вычислительного элемента, который выполняет операцию над данной парой блоков, состоит в следующем - необходимо из битового представления номера одного из блоков, составляющих пару, вычеркнуть бит, расположенный в позиции с номером, равным номеру итерации. Для пояснения описанного алгоритма на рис 9.9 приведен пример разделения блоков данных на пары и представлен механизм выбора вычислительного элемента в случае, когда  $p=4$  (т.е., количество блоков данных равно  $2p=8$ ).

Направление обмена	Пары блоков данных	Номер вычислительного элемента
<b>Итерация 0</b>		
	0 (000) ↔ 4 (100)	0
	1 (001) ↔ 5 (101)	1
	2 (010) ↔ 6 (110)	2
	3 (011) ↔ 7 (111)	3
<b>Итерация 1</b>		
	0 (000) ↔ 2 (010)	0
	1 (001) ↔ 3 (011)	1
	4 (100) ↔ 6 (110)	2
	5 (101) ↔ 7 (111)	3
...		

Рис. 9.9. Разделение блоков данных на пары и определение номера вычислительного элемента, который должен выполнить операцию «Сравнить и разделить» для пары блоков

Следуя предложенной схеме, определим функцию *FindMyPair*, которая для каждого вычислительного элемента с номером *ThreadId* определяет номер пары в массиве *BlockPairs*, над которой данный вычислительный элемент должен выполнить операцию «Сравнить и разделить» на данной итерации *Iter* алгоритма Шелла:

```

// Function for determination of the block pair
// Current thread will perform "compare-split" operation for this block pair

```

```

int FindMyPair (int* BlockPairs, int ThreadID, int Iter) {
    int BlockID=0, index, result;
    for (int i=0; i<ThreadNum; i++) {
        BlockID = BlockPairs[2*i];
        if (Iter == 0)
            index = BlockID%(1<<DimSize-Iter-1);
        if ((Iter>0)&&(Iter<DimSize-1))
            index = ((BlockID>>(DimSize-Iter))<<(DimSize-Iter-1)) |
                (BlockID%(1<<(DimSize-Iter-1)));
        if (Iter == DimSize-1)
            index = BlockID>>1;
        if (index == ThreadID) {
            result = i;
            break;
        }
    }
    return result;
}

```

Необходимо отметить, что предложенный способ определения номера вычислительного элемента по индексам блоков, составляющих пару, обладает важным положительным свойством: на соседних итерациях алгоритма Шелла для каждого вычислительного элемента сохраняется один из обрабатываемых блоков. Таким образом, при переходе от одной итерации к другой, нет необходимости считывать из оперативной памяти в кэш вычислительного элемента оба блока, предназначенные к обработке, – достаточно прочитать только один, «новый» для вычислительного элемента блок. Выигрыш от подобного распределения пар по вычислительным элементам можно получить лишь в том случае, когда вычислительная система такова, что процессоры (ядра) имеют раздельный кэш и размер кэша является достаточным для размещения двух блоков одновременно.

Для того, чтобы обеспечить удачное расположение блоков данных в кэш-памяти вычислительных элементов перед началом выполнения итераций алгоритма Шелла, применим обратную циклическую схему распределения блоков между вычислительными элементами при выполнении локальной сортировки блоков. Это значит, что каждый вычислительный элемент выполняет сортировку блоков данных, которые в структуре гиперкуба имеют номера (*ThreadNum+ThreadId*) и *ThreadId* в указанной последовательности.

Итак, функция *ParallelShellSort* выполняет параллельный алгоритм сортировки Шелла.

```

// Function for parallel Shell sorting
void ParallelShellSort(double* pData, int Size) {
    InitializeParallelSections();
    int* Index = new int [2*ThreadNum];
    int* BlockSize = new int [2*ThreadNum];
    int * BlockPairs = new int [2*ThreadNum];

    for (int i=0; i<2*ThreadNum; i++) {
        Index[i] = int((i*Size)/double(2*ThreadNum));
        if (i<2*ThreadNum-1)
            BlockSize[i] = int (((i+1)*Size)/double(2*ThreadNum)) - Index[i];
        else
            BlockSize[i] = Size-Index[i];
    }

    // Local sorting of data blocks (reverse cycle scheme)
#pragma omp parallel
    {
        int BlockID = ReverseGrayCode(ThreadNum+ThreadId, DimSize);
        QuickSorter(pData, Index[BlockID], Index[BlockID]+BlockSize[BlockID]-1);
        BlockID = ReverseGrayCode(ThreadID, DimSize);
    }
}

```

```

        QuickSorter(pData, Index[BlockID], Index[BlockID]+BlockSize[BlockID]-1);
    }

    // Iterations of the Shell method
    for (int Iter=0; (Iter<DimSize) && (!IsSorted(pData, Size)); Iter++) {
        // Block pairs determination
        SetBlockPairs(BlockPairs, Iter);

        // "Compare-split" operation for data blocks
#pragma omp parallel
    {
        int MyPairNum = FindMyPair(BlockPairs, ThreadID, Iter);
        int FirstBlock = ReverseGrayCode(BlockPairs[2*MyPairNum], DimSize);
        int SecondBlock = ReverseGrayCode(BlockPairs[2*MyPairNum+1], DimSize);
        CompareSplitBlocks(pData, Index[FirstBlock], BlockSize[FirstBlock],
                           Index[SecondBlock], BlockSize[SecondBlock]);
    } // pragma omp parallel
} // for

// Odd-even blocks' transposition
int Iter = 1;
while (!IsSorted(pData, Size)) {
#pragma omp parallel
{
    if (Iter%2 == 0) // Even iteration
        MergeBlocks(pData, Index[2*ThreadID], BlockSize[2*ThreadID],
                     Index[2*ThreadID+1], BlockSize[2*ThreadID+1]);
    else // Odd iteration
        if (ThreadID<ThreadNum-1)
            MergeBlocks(pData, Index[2*ThreadID+1], BlockSize[2*ThreadID+1],
                         Index[2*ThreadID+2], BlockSize[2*ThreadID+2]);
} // pragma omp parallel
Iter++;
} // while

delete [] Index;
delete [] BlockSize;
delete [] BlockPairs;
}

```

### 9.3.5. Результаты вычислительных экспериментов

Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1,86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 9.9. Времена выполнения алгоритмов указаны в секундах.

**Таблица 9.9.** Результаты вычислительных экспериментов для параллельного метода сортировки Шелла (при использовании двух и четырех вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
10000	0,0029	0,0017	1,6638	0,0015	1,8819
20000	0,0062	0,0038	1,6201	0,0030	2,0523
30000	0,0100	0,0059	1,6987	0,0047	2,1254
40000	0,0133	0,0080	1,6652	0,0065	2,0506
50000	0,0172	0,0101	1,7062	0,0080	2,1495

60000	0,0209	0,0124	1,6845	0,0096	2,1687
70000	0,0245	0,0148	1,6556	0,0116	2,1099
80000	0,0278	0,0170	1,6309	0,0140	1,9801
90000	0,0318	0,0200	1,5903	0,0151	2,1011
100000	0,0366	0,0222	1,6469	0,0167	2,1920

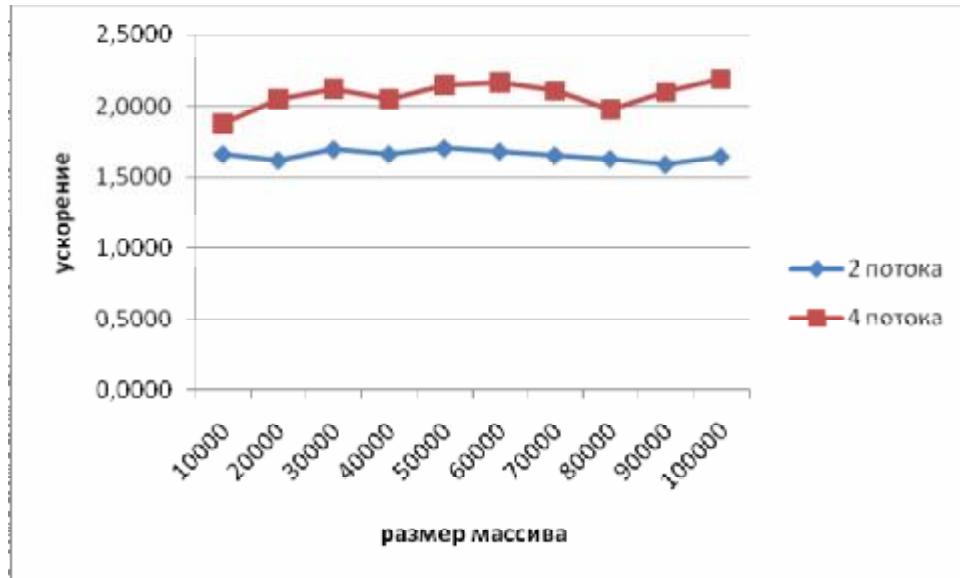


Рис. 9.10. Зависимость ускорения от количества исходных данных при выполнении параллельного метода сортировки Шелла

В таблицах 9.10 и 9.11 и на рис. 9.11 и 9.12 представлены результаты сравнения времени выполнения  $T_p$  параллельного метода сортировки Шелла с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (9.19). Для количества итераций на стадии чет-нечетной перестановки  $L$  использовалось максимальное возможное значение равное  $2p$ . Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0985, а для четырех потоков значение этой величины было оценено как 0,0164.

Как и ранее, латентность  $\alpha$  и пропускная способность канала доступа к оперативной памяти  $\beta$  являются равными  $\alpha = 8,31$  нс и  $\beta = 12,44$  Гб/с. Время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

**Таблица 9.10.** Сравнение экспериментального и теоретического времени выполнения параллельного метода сортировки Шелла с использованием двух потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 9.18 – оценка сверху		Модель 9.19 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
10000	0,0017	0,0021	0,0027	0,0048	0,0003	0,0023
20000	0,0038	0,0044	0,0058	0,0103	0,0006	0,0050
30000	0,0059	0,0069	0,0090	0,0159	0,0009	0,0078
40000	0,0080	0,0094	0,0124	0,0218	0,0012	0,0106
50000	0,0101	0,0120	0,0158	0,0278	0,0016	0,0136
60000	0,0124	0,0146	0,0192	0,0338	0,0019	0,0165
70000	0,0148	0,0173	0,0227	0,0400	0,0022	0,0195

80000	0,0170	0,0200	0,0262	0,0462	0,0026	0,0226
90000	0,0200	0,0227	0,0298	0,0525	0,0029	0,0256
100000	0,0222	0,0254	0,0333	0,0588	0,0033	0,02871

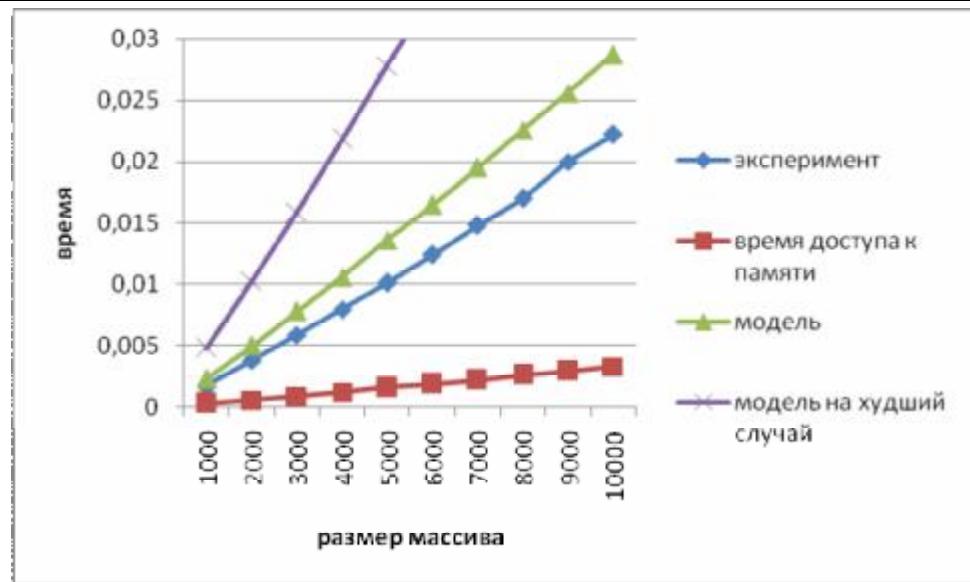


Рис. 9.11. График зависимости экспериментального и теоретического времени выполнения параллельного метода сортировки Шелла от объема исходных данных при использовании двух потоков

**Таблица 9.11.** Сравнение экспериментального и теоретического времени выполнения параллельного метода сортировки Шелла с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 9.18 – оценка сверху		Модель 9.19 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
10000	0,0015	0,0012	0,0032	0,0044	0,0001	0,0013
20000	0,0030	0,0026	0,0068	0,0093	0,0001	0,0027
30000	0,0047	0,0040	0,0105	0,0145	0,0002	0,0042
40000	0,0065	0,0054	0,0143	0,0197	0,0002	0,0057
50000	0,0080	0,0069	0,0181	0,0250	0,0003	0,0072
60000	0,0096	0,0084	0,0220	0,0304	0,0004	0,0088
70000	0,0116	0,0099	0,0260	0,0359	0,0004	0,0103
80000	0,0140	0,0114	0,0300	0,0414	0,0005	0,0119
90000	0,0151	0,0130	0,0340	0,0470	0,0006	0,0135
100000	0,0167	0,0145	0,0381	0,0526	0,0006	0,01514

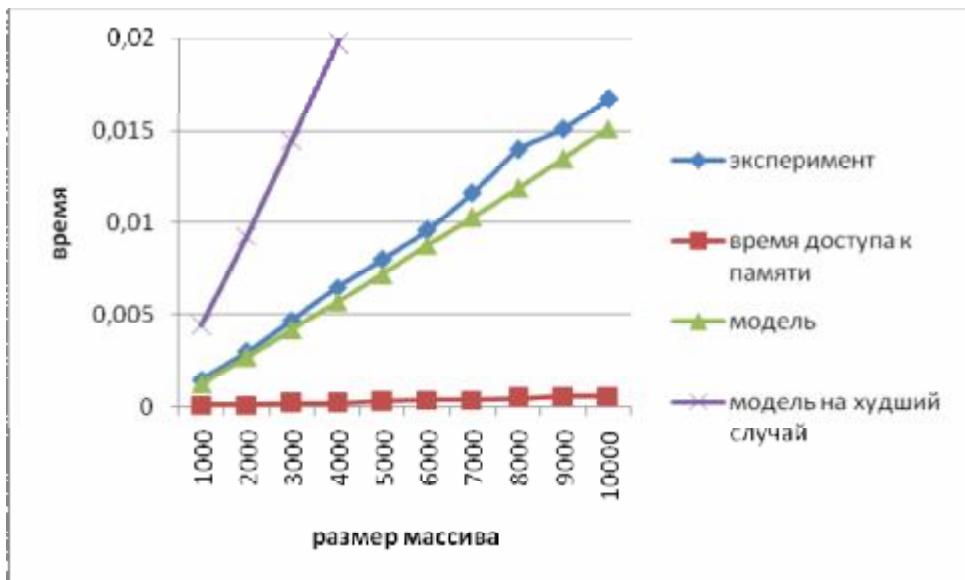


Рис. 9.12. График зависимости экспериментального и теоретического времени выполнения параллельного метода сортировки Шелла от объема исходных данных при использовании четырех потоков

## 9.4. Быстрая сортировка

Алгоритм быстрой сортировки, предложенной Хоаром (*Hoare C.A.R.*), относится к числу эффективных методов упорядочивания данных и широко используется в практических приложениях.

### 9.4.1. Последовательный алгоритм быстрой сортировки

**1. Общая схема метода.** Алгоритма быстрой сортировки основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности (для любой пары блоков все значения одного из этих блоков не превышают значений другого блока). На первой итерации метода осуществляется деление исходного набора данных на первые две части – для организации такого деления выбирается некоторый *ведущий элемент* и все значения набора, меньшие ведущего элемента, переносятся в первый формируемый блок, все остальные значения образуют второй блок набора. На второй итерации сортировки описанные правила применяются рекурсивно для обоих сформированных блоков и т.д. При надлежащем выборе ведущих элементов после выполнения  $\log_2 n$  итераций исходный массив данных оказывается упорядоченным. Более подробное изложение метода может быть получено, например, в [17, 71].

**2. Анализ эффективности.** Эффективность быстрой сортировки в значительной степени определяется правильностью выбора ведущих элементов при формировании блоков. В худшем случае трудоемкость метода имеет тот же порядок сложности, что и пузырьковая сортировка (т.е.  $T_1 \sim n^2$ ). При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма совпадает с быстродействием наиболее эффективных способов сортировки ( $T_1 \sim n \log_2 n$ ). В среднем случае время выполнения алгоритма быстрой сортировки, определяется выражением (см., например, [17, 71]):

$$T_{\text{calc}} = 1,4 \cdot n \log_2 n \cdot t . \quad (9.20)$$

Если размер сортируемого массива настолько велик, что массив не может быть полностью помещен в кэш вычислительного элемента, то по мере выполнения

последовательного алгоритма быстрой сортировки будет происходить чтение данных из оперативной памяти в кэш. Количество чтений определяется порядком выполнения итераций алгоритма и разницей между объемом данных для сортировки и размером кэш памяти. При построении оценки сверху будем считать, что необходимо выполнить чтение всего сортируемого массива из оперативной памяти в кэш на каждой итерации алгоритма быстрой сортировки. Таким образом, время на обращение к оперативной памяти составляет:

$$T_{mem} = 1,4 \cdot \log_2 n \cdot (64n/b). \quad (9.21)$$

Если, как и в предыдущих разделах, учесть латентность памяти, то время доступа к памяти можно подсчитать по следующей формуле:

$$T_{mem} = 1,4 \cdot \log_2 n \cdot n \cdot (a + 64/b). \quad (9.22)$$

Таким образом, общее время выполнения последовательного алгоритма быстрой сортировки может быть определено при помощи выражения:

$$T_1 = 1,4 \cdot n \log_2 n \cdot t + 1,4 \cdot \log_2 n \cdot n(a + 64/b) = 1,4 \cdot n \log_2 n \cdot (t + (a + 64/b)) \quad (9.23)$$

Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см п. 6.5.4):

$$T_1 = 1,4 \cdot n \log_2 n \cdot (t + g(a + 64/b)) \quad (9.24)$$

**3. Программная реализация.** Приведем код рекурсивной функции, выполняющей последовательный алгоритм быстрой сортировки. В качестве ведущего элемента выбирается первый элемент упорядочиваемого набора данных.

```
// Function for serial quick sorting
void SerialQuickSort (double* pData, int first, int last) {
    if (first >= last)
        return;
    int PivotPos = first;
    double Pivot = pData[first];
    for (int i=first+1; i<=last; i++) {
        if (pData[i] < Pivot) {
            if (i != PivotPos+1)
                swap(pData[i], pData[PivotPos+1]);
            PivotPos++;
        }
    }
    swap (pData[first], pData[PivotPos]);
    QuickSorter(pData, first, PivotPos-1);
    QuickSorter(pData, PivotPos+1, last);
}
```

**4. Результаты вычислительных экспериментов.** Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

В таблице 9.12 и на рис. 9.13 представлены результаты сравнения времени выполнения  $T_1$  последовательного алгоритма быстрой сортировки со временем  $T_1^*$ , полученным при помощи модели (9.24). Частота кэш промахов, измеренная с помощью системы VPS, для одного потока значение этой величины было оценено как 0,0057.

**Таблица 9.12.** Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма быстрой сортировки

Размер матриц	$T_1$	$T_1^*$ (calc) (модель)	Модель 9.23 – оценка сверху		Модель 9.24 – уточненная оценка	
			$T_1^*$ (mem)	$T_1^*$	$T_1^*$ (mem)	$T_1^*$

10000	0,0029	0,0037	0,0024	0,0062	0,0000	0,0037
20000	0,0062	0,0080	0,0052	0,0132	0,0000	0,0080
30000	0,0100	0,0125	0,0082	0,0207	0,0000	0,0125
40000	0,0133	0,0171	0,0112	0,0283	0,0001	0,0172
50000	0,0172	0,0218	0,0143	0,0361	0,0001	0,0219
60000	0,0209	0,0266	0,0175	0,0441	0,0001	0,0267
70000	0,0245	0,0315	0,0207	0,0522	0,0001	0,0316
80000	0,0278	0,0364	0,0239	0,0603	0,0001	0,0366
90000	0,0318	0,0414	0,0272	0,0686	0,0002	0,0416
100000	0,0366	0,0464	0,0305	0,0769	0,0002	0,04662



Рис. 9.13. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма быстрой сортировки от объема исходных данных

#### 9.4.2. Параллельный алгоритм быстрой сортировки

**1. Организация параллельных вычислений.** Параллельное обобщение алгоритма быстрой сортировки (см., например, [85]) наиболее простым способом может быть получено для случая, когда потоки параллельной программы могут быть организованы в виде  $N$ -мерного гиперкуба (т.е. количество вычислительных элементов  $p=2^N$ ). Пусть, как и ранее, исходный набор данных логически разделен на  $2p$  блоков одинакового размера  $n/2p$ . Тогда блоки данных образуют  $(N+1)$ -мерный гиперкуб. Возможный способ выполнения первой итерации параллельного метода при таких условиях может состоять в следующем:

- выбрать каким-либо образом ведущий элемент (например, в качестве ведущего элемента можно взять среднее арифметическое элементов, расположенных на выбранном ведущем блоке);
- сформировать пары блоков, для которых необходимо выполнить взаимообмен данными на данной итерации алгоритма: пары образуют блоки, для которых битовое представление номеров отличается только в позиции  $(N+1)$ ;
- для каждой пары блоков определить вычислительный элемент, который будет выполнять необходимые операции (для определения номера вычислительного элемента

по индексам блоков, составляющих пару, можно воспользоваться алгоритмом, предложенным при рассмотрении параллельного варианта метода Шелла);

- параллельно выполнить операцию «Сравнить и разделить» над всеми парами блоков, в результате такого обмена в блоках, для которых в битовом представлении номера бит позиции  $N+1$  равен 0, должны оказаться части блоков со значениями, меньшими ведущего элемента; блоки с номерами, в которых бит  $N+1$  равен 1, должны собрать, соответственно, все значения данных, превышающие значение ведущего элемента.

В результате выполнения такой итерации сортировки исходный набор оказывается разделенным на две части, одна из которых (со значениями меньшими, чем значение ведущего элемента) располагается в блоках данных, в битовом представлении номеров которых бит  $N+1$  равен 0. Таких блоков всего  $p$  и, таким образом, исходный  $(N+1)$ -мерный гиперкуб также оказывается разделенным на два гиперкуба размерности  $N$ . К этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После  $(N+1)$ -кратного повторения подобных итераций для завершения сортировки достаточно упорядочить полученные блоки данных, каждый вычислительный элемент упорядочивает 2 блока.

Для пояснения на рис.9.14 представлен пример упорядочивания данных при  $n=16$ ,  $p=2$  (т.е. исходный массив разбит на  $2p = 4$  блока, каждый блок данных содержит 4 значения). На этом рисунке блоки данных изображены в виде прямоугольников; значения блоков приводятся в начале и при завершении каждой итерации сортировки. Взаимодействующие пары блоков соединены двунаправленными стрелками. Для разделения данных выбирались наилучшие значения ведущих элементов: на первой итерации для всех блоков использовалось значение 0, на второй итерации для пары блоков 0, 1 ведущий элемент равен -5, для пары блоков 2, 3 это значение было принято равным 4.

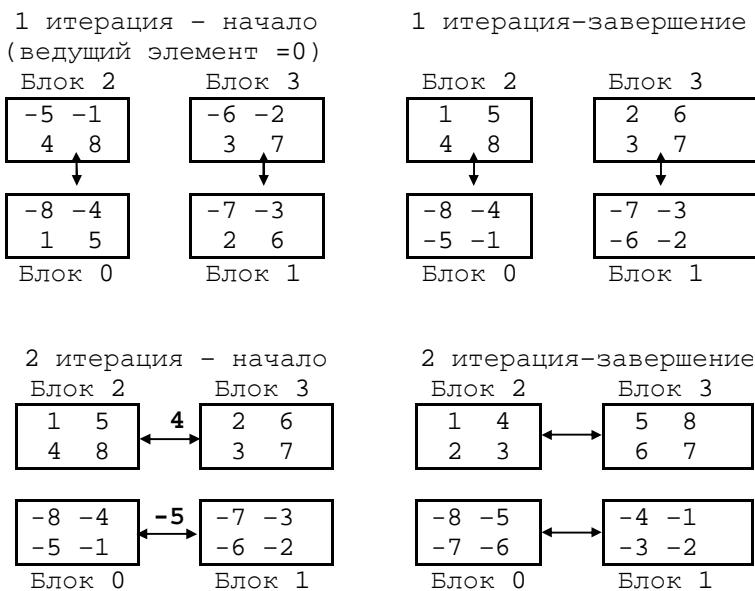


Рис. 9.14. Пример упорядочивания данных параллельным методом быстрой сортировки (без результатов локальной сортировки блоков)

**2. Анализ эффективности.** Оценим трудоемкость рассмотренного параллельного метода. Пусть у нас имеется  $(N+1)$ -мерный гиперкуб, состоящий из  $2p=2^{N+1}$  блоков данных, где  $2p < n$ .

Эффективность параллельного метода быстрой сортировки, как и в последовательном варианте, во многом зависит от правильности выбора значений ведущих элементов.

Определение общего правила для выбора этих значений представляется затруднительным. Сложность такого выбора может быть снижена, если выполнить упорядочение локальных блоков процессоров перед началом сортировки и обеспечить однородное распределение сортируемых данных между потоками параллельной программы.

Определим вначале вычислительную сложность алгоритма сортировки. На каждой из  $\log_2(2p)$  итераций сортировки каждый поток осуществляет операцию «Сравнить и разделить» над парой блоков в соответствии со значением ведущего элемента. Сложность этой операции составляет  $n/p$  операций (будем предполагать, что на каждой итерации блок данных делится на равные по размеру части относительно ведущего элемента и, следовательно, размер блоков данных в процессе сортировки остается постоянным).

При завершении вычислений потоки выполняют сортировку двух блоков, что может быть выполнено при использовании быстрых алгоритмов за

$$2 \cdot 1,4 \cdot (n/2p) \log_2(n/2p)$$

операций.

Таким образом, общее время вычислений параллельного алгоритма быстрой сортировки составляет

$$T_p(\text{calc}) = [(n/p) \log_2(2p) + 2 \cdot 1,4(n/2p) \log_2(n/2p)] \cdot t,$$

где  $t$  есть время выполнения базовой операции перестановки.

Предположим, что выбор ведущих элементов осуществляется самым наилучшим образом, количество итераций алгоритма равно  $\log_2(2p)$ , и все блоки данных сохраняют постоянный размер  $(n/2p)$ . При таких условиях, на каждой итерации сравнения блоков выполняется считывание в кэш из оперативной памяти всего сортируемого массива. Далее на этапе локальной сортировки блоков каждый вычислительный элемент считывает упорядочиваемые блоки данных на каждой итерации повторно (см. оценку (9.14)). Таким образом, затраты на считывание необходимых данных из оперативной памяти в кэш составляют:

$$T_p(\text{mem}) = \log_2(2p) \cdot \frac{64n}{b} + p \cdot 2 \cdot 1,4 \log_2(n/2p) \cdot \frac{64(n/2p)}{b} = (\log_2(2p) + 1,4 \log_2(n/2p)) \cdot \frac{64n}{b}, \quad (9.25)$$

Если, как и в предыдущих разделах, учесть латентность памяти, то время доступа к памяти можно подсчитать по следующей формуле:

$$T_p(\text{mem}) = (\log_2(2p) + 1,4 \log_2(n/2p)) \cdot n(a + 64/b) \quad (9.26)$$

С учетом всех полученных соотношений общая трудоемкость алгоритма оказывается равной:

$$T_p = [(n/p) \log_2(2p) + 2 \cdot 1,4(n/2p) \log_2(n/2p)]t + (\log_2(2p) + 1,4 \log_2(n/2p)) \cdot n(a + 64/b). \quad (9.27)$$

Кроме того, необходимо учесть накладные расходы на организацию параллельности. На каждой итерации алгоритма создается параллельная секция для выполнения операции «Сравнить и разделить». Еще одна параллельная секция создается для выполнения финальной локальной сортировки блоков на всех потоках параллельной программы, т.е.:

$$T_p = [(n/p) \log_2(2p) + 2 \cdot 1,4(n/2p) \log_2(n/2p)]t + (\log_2(2p) + 1,4 \log_2(n/2p)) \cdot n(a + 64/b) + \\ + (\log_2(2p) + 1) \cdot d \quad (9.28)$$

Для более точной оценки необходимо учесть частоту кэш промахов  $g$ :

$$T_p = [(n/p) \log_2(2p) + 2 \cdot 1,4(n/2p) \log_2(n/2p)]t + g(\log_2(2p) + 1,4 \log_2(n/2p)) \cdot n(a + 64/b) + \\ + (\log_2(2p) + 1) \cdot d \quad (9.29)$$

Следует отметить, что при построении оценок предполагалось, что выбор ведущих элементов осуществляется наилучшим образом. На практике обеспечить такой выбор ведущих элементов, который приводил бы к равному разделению блоков потоков и, соответственно, к равномерному распределению вычислительной нагрузки, достаточно сложно.

**3. Программная реализация.** Рассмотрим возможный вариант реализации параллельного варианта метода быстрой сортировки. Как и ранее, объявим несколько глобальных переменных: *ThreadNum* для определения количества потоков в параллельной программе, *DimSize* для определения размерности гиперкуба, который может быть составлен из заданного количества блоков данных, *ThreadID* для определения номера текущего потока. Создадим локальные копии переменной *ThreadID* при помощи директивы *threadprivate*. Функция *InitializeParallelSections* определяет количество потоков *ThreadNum* и размерность виртуального гиперкуба *DimSize*, а также идентификатор потока *ThreadID*.

Алгоритм выбора ведущих элементов для выполнения операции сравнения и разделения блоков основан на информации о распределении сортируемых значений. Если минимальное возможное значение элемента сортируемого набора равно *MIN*, а максимальное – *MAX*, то в качестве ведущего элемента на первой итерации алгоритма выбирается среднее арифметическое значение *Pivot* = (*MIN*+*MAX*)/2. Далее, предполагая равномерное изменение значений упорядочиваемых данных, на второй итерации алгоритма для подгиперкуба с меньшими номерами блоков будем использовать значение *Pivot* = (3*MIN*+*MAX*)/4, а для подгиперкуба с большими номерами блоков – *Pivot* = (*MIN*+3*MAX*)/4, и т.д.

Поскольку при выполнении параллельного алгоритма быстрой сортировки достаточно сложно обеспечить идеальный выбор ведущего элемента, блоки данных различных вычислительных элементов могут иметь разный размер. В худшем случае, в какой-то момент времени все данные могут быть сосредоточены в блоке одного вычислительного элемента. Это делает невозможным хранение блоков в рамках исходного массива. Поэтому для раздельного хранения блоков данных разных вычислительных элементов заведем систему буферов *pTempData*, каждый из которых может вместить *Size* элементов. В ходе сортировки упорядочиваемые данные размещаются в этих буферах; количество элементов, размещаемых в *i*-ом блоке *pTempData[i]*, определяется значением переменной *BlockSize[i]*.

После выполнения *DimSize* итераций сравнения и разделения блоков, выполняется локальная сортировка блоков. Далее данные из буферов *pTempData* собираются в исходный массив *pData*. После выполнения указанных действий массив оказывается отсортированным.

Функция *ParallelQuickSort* выполняет параллельный алгоритм быстрой сортировки:

```
// Function for parallel quick sorting
void ParallelQuickSort (double* pData, int Size) {
    InitializeParallelSections();
    double ** pTempData = new double * [2*ThreadNum];
    int * BlockSize = new int [2*ThreadNum];
    double* Pivots = new double [ThreadNum];
    int * BlockPairs = new int [2*ThreadNum];
    for (int i=0; i<2*ThreadNum; i++) {
        pTempData[i] = new double [Size];
        BlockSize[i] = Size/(2*ThreadNum);
    }
    for (int j=0; j<Size; j++)
        pTempData[2*j*ThreadNum/Size][j%(Size/(2*ThreadNum))] = pData[j];

    // Iterations of quick sorting
    for (int i=0; i<DimSize; i++) {
        // Determination of pivot values
        for (int j=0; j<ThreadNum; j++)
            Pivots[j] = (MAX_VALUE + MIN_VALUE)/2;
        for (int iter=1; iter<=i; iter++)
            for (int j=0; j<ThreadNum; j++)
                Pivots[j] = Pivots[j] - pow(-1.0f, j/((2*ThreadNum)>>(iter+1))) *
```

```

        (MAX_VALUE-MIN_VALUE)/(2<<iter);

    // Determination of data block pairs
    SetBlockPairs(BlockPairs, i);

#pragma omp parallel
{
    int MyPair = FindMyPair(BlockPairs, ThreadID, i);
    int FirstBlock = BlockPairs[2*MyPair];
    int SecondBlock = BlockPairs[2*MyPair+1];
    CompareSplitBlocks(pTempData[FirstBlock], BlockSize[FirstBlock],
                       pTempData[SecondBlock], BlockSize[SecondBlock], Pivots[ThreadID]);
} // pragma omp parallel
} // for

// Local sorting
#pragma omp parallel
{
    if (BlockSizes[2*ThreadID]>0)
        SerialQuickSort(pTempData[2*ThreadID], BlockSize[2*ThreadID]);
    if (BlockSizes[2*ThreadID+1]>0)
        SerialQuickSort(pTempData[2*ThreadID+1], BlockSize[2*ThreadID+1]);
}

int curr = 0;
for (int i=0; i<2*ThreadNum; i++)
    for (int j=0; (j<BlockSize[i])&&(curr<Size); j++)
        pData[curr++] = pTempData[i][j];

for (int i=0; i<ThreadNum; i++)
    delete [] pTempData[i];
delete [] pTempData;
delete [] BlockSize;
delete [] Pivots;
delete [] BlockPairs;
}

```

Следует отметить, что при реализации параллельного алгоритма быстрой сортировки пары блоков данных формируются непосредственно на основании индексов этих блоков, в отличие от сортировки Шелла, где пары блоков формировались на основе кодов Грея индексов блоков:

```

// Function for block pairs determination.
// "Compare-split" operation will be carried out for that pairs.
void SetBlockPairs (int* BlockPairs, int Iter) {
    int PairNum = 0, FirstValue, SecondValue;
    bool Exist;
    for (int i=0; i<2*ThreadNum; i++) {
        FirstValue = i;
        Exist = false;
        for (int j=0; (j<PairNum)&&(!Exist); j++)
            if (BlockPairs[2*j+1] == FirstValue)
                Exist = true;
        if (!Exist) {
            SecondValue = FirstValue^(1<<(DimSize-Iter-1));
            BlockPairs[2*PairNum] = FirstValue;
            BlockPairs[2*PairNum+1] = SecondValue;
            PairNum++;
        } // if
    } // for
}

```

Функция *CompareSplitBlocks* выполняет операцию «Сравнить и разделить» для блоков *FirstBlock* и *SecondBlock* указанных размеров в соответствии со значением элемента *Pivot*.

После выполнения операции в блоке *FirstBlock* оказываются значения из обоих блоков, меньшие ведущего элемента, а в блоке *SecondBlock* – значения, большие ведущего элемента:

```
// Function for carrying out the "compare-split" operation
// for two unsorted data blocks according to the pivot value
void CompareSplitBlocks (double* pFirstBlock, int &FirstBlockSize,
    double* pSecondBlock, int &SecondBlockSize, double Pivot) {
    int TotalSize = FirstBlockSize + SecondBlockSize;
    double* pTempBlock = new double [TotalSize];
    int LastMin = 0, FirstMax = TotalSize - 1;
    for (int i=0; i<FirstBlockSize; i++) {
        if (pFirstBlock[i]<Pivot)
            pTempBlock[LastMin++] = pFirstBlock[i];
        else
            pTempBlock[FirstMax--] = pFirstBlock[i];
    }
    for (int i=0; i<SecondBlockSize; i++) {
        if (pSecondBlock[i]<Pivot)
            pTempBlock[LastMin++] = pSecondBlock[i];
        else
            pTempBlock[FirstMax--] = pSecondBlock[i];
    }
    FirstBlockSize = LastMin;
    SecondBlockSize = TotalSize - LastMin;
    for (int i=0; i<FirstBlockSize; i++)
        pFirstBlock[i] = pTempBlock[i];
    for (int i=0; i<SecondBlockSize; i++)
        pSecondBlock[i] = pTempBlock[FirstBlockSize+i];
    delete [] pTempBlock;
}
```

**4. Результаты вычислительных экспериментов.** Эксперименты проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1,86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 9.13. Времена выполнения алгоритмов указаны в секундах.

**Таблица 9.13.** Результаты вычислительных экспериментов для параллельного метода быстрой сортировки (при использовании двух и четырех вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
10000	0,0029	0,0025	1,1411	0,0022	1,3153
20000	0,0062	0,0053	1,1689	0,0043	1,4365
30000	0,0100	0,0079	1,2591	0,0070	1,4265
40000	0,0133	0,0106	1,2511	0,0098	1,3498
50000	0,0172	0,0135	1,2689	0,0117	1,4752
60000	0,0209	0,0163	1,2827	0,0136	1,5360
70000	0,0245	0,0201	1,2178	0,0159	1,5406
80000	0,0278	0,0227	1,2260	0,0183	1,5220
90000	0,0318	0,0253	1,2574	0,0207	1,5371
100000	0,0366	0,0282	1,2972	0,0233	1,5712

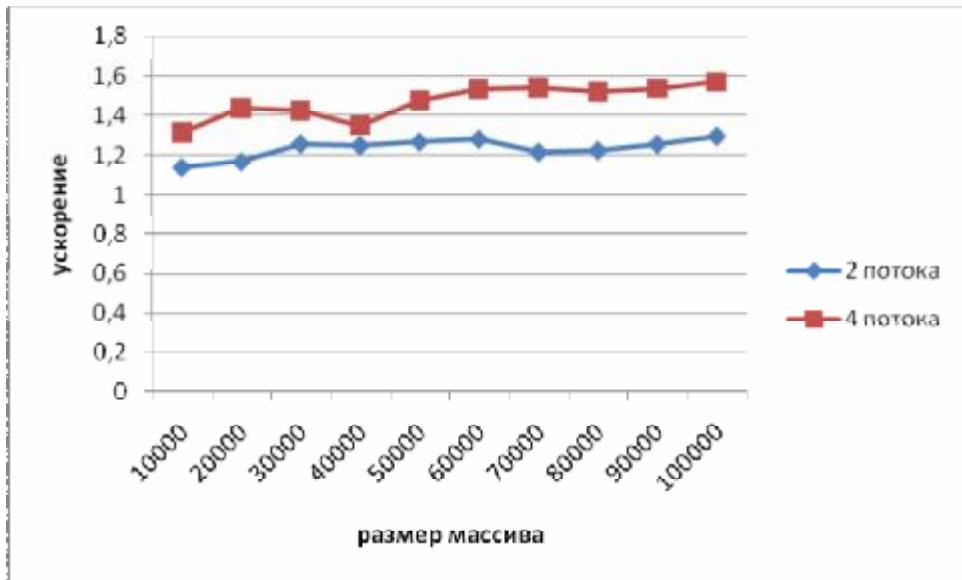


Рис. 9.15. Зависимость ускорения от количества исходных данных при выполнении параллельного метода быстрой сортировки

В таблицах 9.14 и 9.15 и на рис. 9.16 и 9.7 представлены результаты сравнения времени выполнения  $T_p$  параллельного метода быстрой сортировки с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (9.29). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,2098, а для четырех потоков значение этой величины было оценено как 0,4175.

Оценки времени латентности  $\alpha$  и величины пропускной способности канала доступа к оперативной памяти  $\beta$  проводилась в п. 6.5.4 и определены для используемого вычислительного узла как  $\alpha = 8,31$  нс и  $\beta = 12,44$  Гб/с. Как и ранее, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах исходного массива составляет не более 2%.

**Таблица 9.14.** Сравнение экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки с использованием двух потоков

Размер матриц	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 9.28 – оценка сверху		Модель 9.29 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
10000	0,0025	0,0018	0,0023	0,0041	0,0005	0,0023
20000	0,0053	0,0038	0,0050	0,0089	0,0011	0,0049
30000	0,0079	0,0060	0,0079	0,0139	0,0017	0,0077
40000	0,0106	0,0082	0,0108	0,0190	0,0023	0,0105
50000	0,0135	0,0105	0,0138	0,0243	0,0029	0,0134
60000	0,0163	0,0128	0,0168	0,0297	0,0035	0,0164
70000	0,0201	0,0152	0,0199	0,0351	0,0042	0,0194
80000	0,0227	0,0176	0,0231	0,0406	0,0048	0,0224
90000	0,0253	0,0200	0,0262	0,0462	0,0055	0,0255
100000	0,0282	0,0224	0,0294	0,0518	0,0062	0,0286

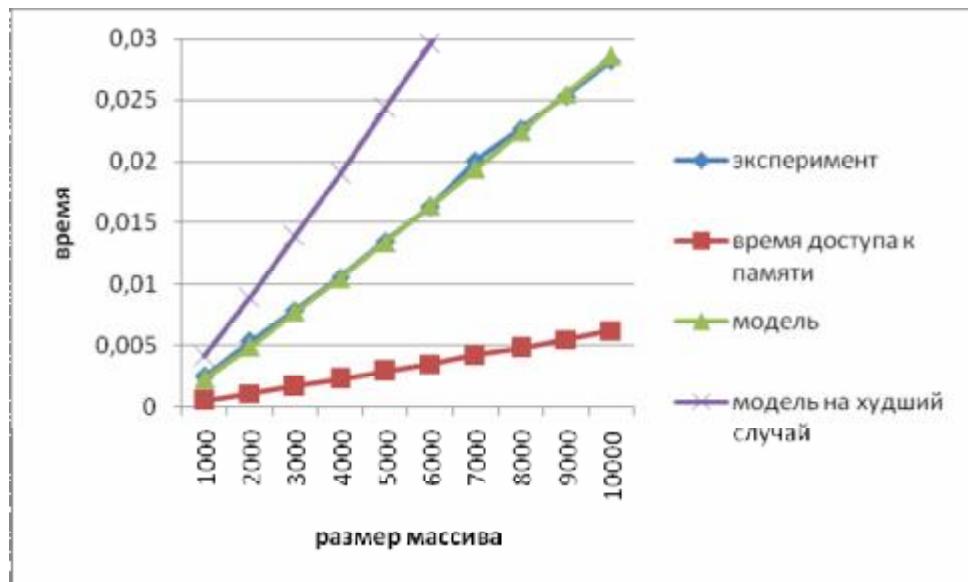


Рис. 9.16. График зависимости экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки от объема исходных данных при использовании двух потоков

**Таблица 9.15.** Сравнение экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 9.28 – оценка сверху		Модель 9.29 – уточненная оценка	
			$T_p^* \text{ (tem)}$	$T_p^*$	$T_p^* \text{ (tem)}$	$T_p^*$
10000	0,0022	0,0009	0,0023	0,0032	0,0010	0,0018
20000	0,0043	0,0019	0,0049	0,0068	0,0021	0,0039
30000	0,0070	0,0029	0,0077	0,0107	0,0032	0,0062
40000	0,0098	0,0040	0,0106	0,0146	0,0044	0,0085
50000	0,0117	0,0052	0,0135	0,0187	0,0056	0,0108
60000	0,0136	0,0063	0,0165	0,0228	0,0069	0,0132
70000	0,0159	0,0075	0,0196	0,0270	0,0082	0,0156
80000	0,0183	0,0086	0,0226	0,0313	0,0095	0,0181
90000	0,0207	0,0098	0,0258	0,0356	0,0108	0,0206
100000	0,0233	0,0110	0,0289	0,0399	0,0121	0,02308

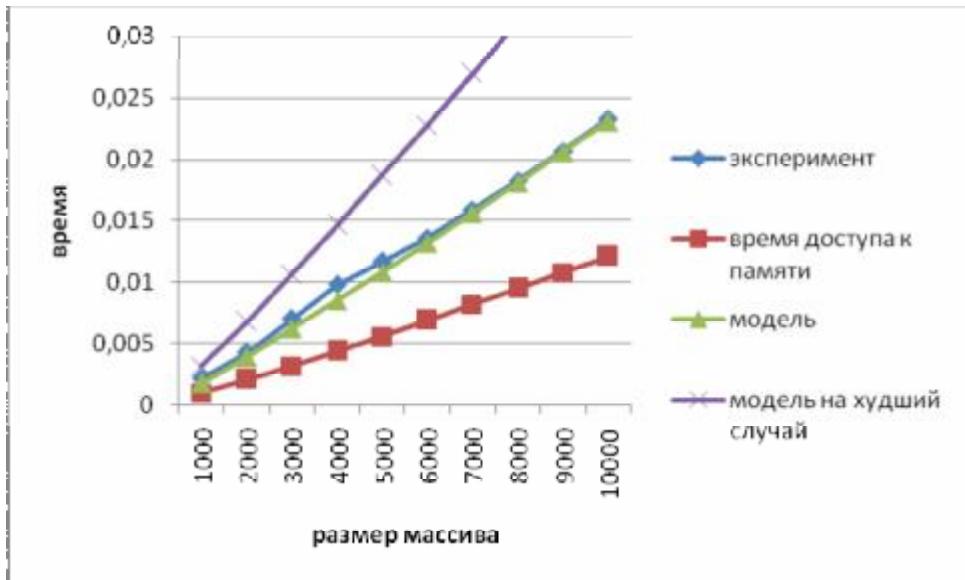


Рис. 9.17. График зависимости экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки от объема исходных данных при использовании четырех потоков

#### 9.4.3. Обобщенный алгоритм быстрой сортировки

В обобщенном алгоритме быстрой сортировки (*HyperQuickSort algorithm*) в дополнение к обычному методу быстрой сортировки предлагается конкретный способ выбора ведущих элементов. Суть предложения состоит в том, что сортировка блоков данных выполняется в самом начале выполнения вычислений. Кроме того, для поддержки упорядоченности в ходе вычислений над блоками данных выполняется операция слияния, а затем деление полученного блока двойного размера согласно ведущему элементу. Как результат, в силу упорядоченности блоков, при выполнении алгоритма быстрой сортировки в качестве ведущего элемента целесообразнее будет выбирать средний элемент какого-либо блока. Выбираемый подобным образом ведущий элемент в отдельных случаях может оказаться более близок к реальному среднему значению всего сортируемого набора, чем какое-либо другое произвольно выбранное значение.

Все остальные действия в новом рассматриваемом алгоритме выполняются в соответствии с обычным методом быстрой сортировки. Более подробное описание данного способа распараллеливания быстрой сортировки может быть получено, например, в [85].

**1. Анализ эффективности.** При анализе эффективности обобщенного алгоритма можно воспользоваться соотношением (9.29). Следует только учесть, что на каждой итерации метода теперь выполняется операция слияния блоков (будем, как и ранее, предполагать, что их размер одинаков и равен  $(n/2p)$ ). Кроме того, на каждой итерации метода создаются две параллельные секции: одна для выбора ведущих элементов, вторая для выполнения слияния блоков. С учетом всех высказанных замечаний трудоемкость обобщенного алгоритма быстрой сортировки может быть выражена при помощи соотношения следующего вида:

$$T_p = [(n/p) \log_2(2p) + 2 \cdot 1,4(n/2p) \log_2(n/2p)]t + (\log_2(2p) + 1,4 \log_2(n/2p)) \cdot n(a + 64/b) + (2(\log_2 p + 1) + 1) \cdot d \quad (9.30)$$

Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см п. 6.5.4):

$$T_p = [(n/p) \log_2(2p) + 2 \cdot 1,4(n/2p) \log_2(n/2p)]t + g(\log_2(2p) + 1,4 \log_2(n/2p)) \cdot n(a + 64/b) + (2(\log_2 p + 1) + 1) \cdot d \quad (9.31)$$

**2. Программная реализация.** Представим возможный вариант параллельной программы обобщенной быстрой сортировки. При этом реализация отдельных модулей не

приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

Как и ранее, введем глобальные переменные: *ThreadNum* для определения количества потоков в параллельной программе, *DimSize* для определения размерности гиперкуба, который может быть составлен из заданного количества блоков данных, *ThreadID* для определения номера текущего потока, *PairNum*, *FirstBlock* и *SecondBlock* для определения номера пары блоков, которую данный поток должен обработать на данной итерации, а также конкретных номеров блоков данных потока в структуре виртуального гиперкуба, *GroupID* для определения номера группы блоков, к которой принадлежит данный данный пара на текущей итерации обмена данными (номер группы блоков позволяет определить индекс ведущего элемента, который данный поток должен использовать для операции сравнения и разделения блоков на текущей итерации алгоритма). На первой итерации все блоки входят в одну группу, номер этой группы равен 0, на второй итерации происходит разделение исходного виртуального гиперкуба на два подгиперкуба меньшей размерности. При этом половина блоков (блоки, в битовом представлении номеров которых старший бит равен 0) формируют группу с номером 0, а другая половина блоков – группу с номером 1, и т.д. Создадим локальные копии переменных, значения которых различаются в разных потоках, при помощи директивы *threadprivate*.

```

int ThreadNum; // Number of threads
int ThreadID; // Thread identifier
int DimSize; // Number of dimension in hypercube, assembled of data blocks
int MyPair; // Number of the block pair, that should be processed
            // by current thread
int FirstBlock; // Number of the first data block in pair
int SecondBlock; // Number of the second data block in pair
int GroupID; // Number of the group current thread belongs to

#pragma omp threadprivate (ThreadID, MyPair, FirstBlock,
                         SecondBlock, GroupID)

```

Необходимо пояснить схему выбора ведущих элементов. Один из блоков каждой группы должен задать значение ведущего элемента, который будет использоваться для обработки всех блоков данной группы. Как описано выше, в качестве ведущего значения выбирается средний элемент блока. Для выбора ведущих элементов на каждой итерации алгоритма организуется цикл по количеству групп (на первой итерации алгоритма все блоки данных входят в одну группу, на второй итерации общее количество блоков разделяется на 2 группы, на третьей итерации – 4 группы и так далее). На каждой итерации этого цикла задается значение ведущего элемента для одной группы – в качестве такого элемента выбирается среднее значение блока с минимальным номером, входящего в группу.

```

// Function for parallel hyperquick sorting
void ParallelHyperQuickSort (double* pData, int Size) {
    InitializeParallelSections();
    double ** pTempData = new double * [2*ThreadNum];
    int * BlockSizes = new int [2*ThreadNum];
    double* Pivots = new double [ThreadNum];
    int * BlockPairs = new int [2*ThreadNum];
    for (int i=0; i<2*ThreadNum; i++) {
        pTempData[i] = new double [Size];
        for (int j=0; j<Size; j++)
            pTempData[i][j] = DUMMY_VALUE;
        BlockSizes[i] = Size/(2*ThreadNum);
    }
    for (int j=0; j<Size; j++)
        pTempData[2*j*ThreadNum/Size][j%(Size/(2*ThreadNum))] = pData[j];

    // Local sorting of data blocks

```

```

#pragma omp parallel
{
    LocalQuickSort(pTempData[ThreadNum+ThreadID], 0,
                   BlockSizes[ThreadNum+ThreadID]-1);
    LocalQuickSort(pTempData[ThreadID], 0, BlockSizes[ThreadID]-1);
}

// Iterations of parallel hyperquick sorting algorithm
for (int i=0; i<DimSize; i++) {
    // Determination of data block pairs
    SetBlockPairs(BlockPairs, i);

#pragma omp parallel for
    for (int j=0; j<(1<<i); j++) {
        int BlockID = (2*ThreadNum*j)/(1<<i);
        Pivots[j] = pTempData[BlockID][BlockSizes[BlockID]/2];
    } // pragma omp parallel for

    // Carrying out the "compare-split" operation for sorted data blocks
#pragma omp parallel
{
    MyPair = FindMyPair(BlockPairs, ThreadID, i);
    FirstBlock = BlockPairs[2*MyPair];
    SecondBlock = BlockPairs[2*MyPair+1];
    GroupID = FirstBlock/(1<<(DimSize-i));
    CompareSplitBlocks(pTempData[FirstBlock], BlockSizes[FirstBlock],
                       pTempData[SecondBlock], BlockSizes[SecondBlock], Pivots[GroupID]);
}
} // for

int curr = 0;
for (int i=0; i<2*ThreadNum; i++)
    for (int j=0; j<BlockSizes[i]; j++)
        pData[curr++] = pTempData[i][j];

for (int i=0; i<2*ThreadNum; i++)
    delete [] pTempData[i];

delete [] pTempData;
delete [] BlockSizes;
delete [] Pivots;
delete [] BlockPairs;
}

```

Функция *CompareSplitBlocks* в данном случае выполняет слияние и разделение двух упорядоченных блоков:

```

// Function for carrying out the "compare-split" operation
// for two sorted data blocks according to the pivot value
void CompareSplitBlocks(double* pFirstBlock, int &FirstBlockSize,
                       double* pSecondBlock, int &SecondBlockSize, double Pivot) {
    int TotalSize = FirstBlockSize + SecondBlockSize;
    double* TempBlock = new double [TotalSize];
    int i=0, j=0, curr=0;
    while ((i<FirstBlockSize)&&(j<SecondBlockSize)) {
        if (pFirstBlock[i]<pSecondBlock[j])
            TempBlock[curr++] = pFirstBlock[i++];
        else
            TempBlock[curr++] = pSecondBlock[j++];
    }
    while (i<FirstBlockSize)
        TempBlock[curr++] = pFirstBlock[i++];
    while (j<SecondBlockSize)
        TempBlock[curr++] = pSecondBlock[j++];
}

```

```

curr = 0;
while ((curr<TotalSize) && (TempBlock[curr]<Pivot))
    pFirstBlock[curr] = TempBlock[curr++];
FirstBlockSize = curr;
SecondBlockSize = TotalSize - curr;
while (curr<TotalSize)
    pSecondBlock[curr-FirstBlockSize] = TempBlock[curr++];
delete [] TempBlock;
}

```

**3. Результаты вычислительных экспериментов.** Вычислительные эксперименты для оценки эффективности параллельного варианта метода обобщенной быстрой сортировки проводились при условиях, указанных в п. 9.2.1. Вычисления проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1,86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Результаты вычислительных экспериментов приведены в таблице 9.16. Времена выполнения алгоритмов указаны в секундах.

**Таблица 9.16.** Результаты вычислительных экспериментов для параллельного метода обобщенной быстрой сортировки (при использовании двух и четырех вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
10000	0,0029	0,0022	1,3075	0,0018	1,5793
20000	0,0062	0,0046	1,1689	0,0037	1,4365
30000	0,0100	0,0071	1,2591	0,0062	1,4265
40000	0,0133	0,0099	1,2511	0,0082	1,3498
50000	0,0172	0,0127	1,2689	0,0106	1,4752
60000	0,0209	0,0154	1,2827	0,0131	1,5360
70000	0,0245	0,0184	1,2178	0,0150	1,5406
80000	0,0278	0,0334	1,2260	0,0176	1,5220
90000	0,0318	0,0239	1,2574	0,0198	1,5371
100000	0,0366	0,0273	1,2972	0,0224	1,5712

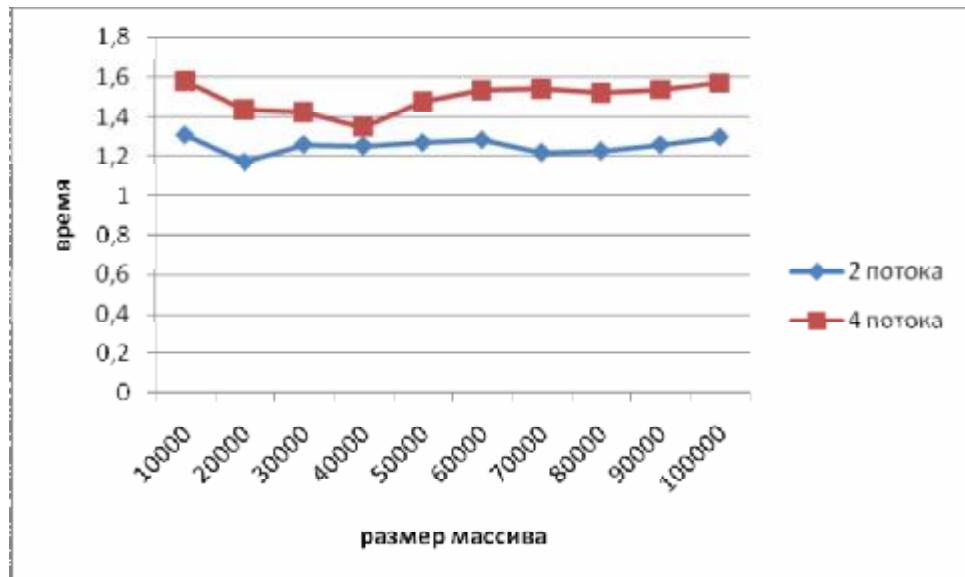
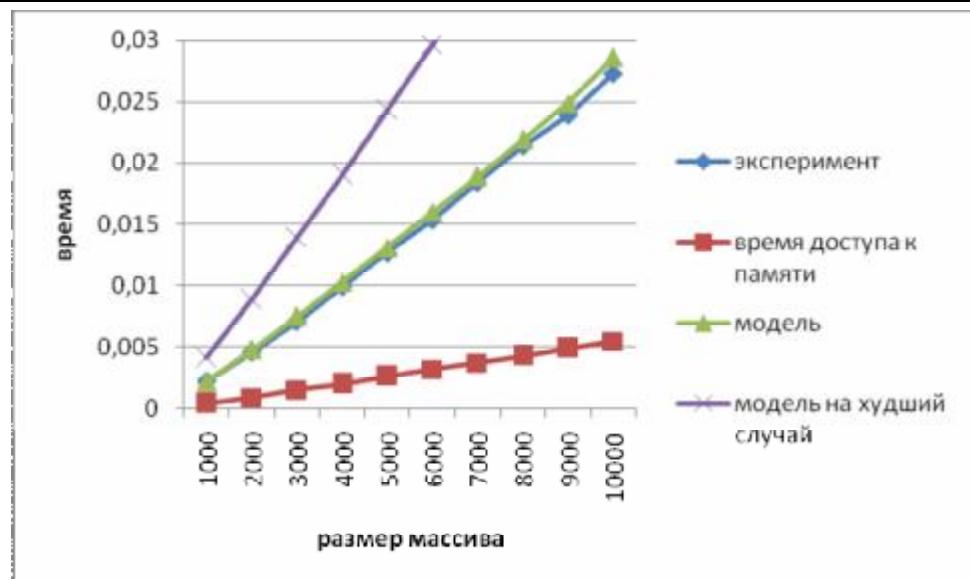


Рис. 9.18. Зависимость ускорения от количества исходных данных при выполнении параллельного метода обобщенной быстрой сортировки

В таблицах 9.17 и 9.18 и на рис. 9.19 и 9.20 представлены результаты сравнения времени выполнения  $T_p$  параллельного метода обобщенной быстрой сортировки с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (9.31). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,188, а для четырех потоков значение этой величины было оценено как 0,247. Как и ранее, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

**Таблица 9.17.** Сравнение экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки с использованием двух потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 9.30 – оценка сверху		Модель 9.31 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
10000	0,0022	0,0018	0,0023	0,0041	0,0004	0,0022
20000	0,0046	0,0038	0,0050	0,0089	0,0009	0,0048
30000	0,0071	0,0060	0,0079	0,0139	0,0015	0,0075
40000	0,0099	0,0082	0,0108	0,0190	0,0020	0,0103
50000	0,0127	0,0105	0,0138	0,0243	0,0026	0,0131
60000	0,0154	0,0128	0,0168	0,0297	0,0032	0,0160
70000	0,0184	0,0152	0,0199	0,0351	0,0037	0,0189
80000	0,0214	0,0176	0,0231	0,0406	0,0043	0,0219
90000	0,0239	0,0200	0,0262	0,0462	0,0049	0,0249
100000	0,0273	0,0224	0,0294	0,0518	0,0055	0,0280



**Рис. 9.19.** График зависимости экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки от объема исходных данных при использовании двух потоков

**Таблица 9.18.** Сравнение экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки с использованием четырех потоков

Размер	$T_p$	$T_p^* (calc)$	Модель 9.30	Модель 9.31
--------	-------	----------------	-------------	-------------

матриц		(модель)	оценка сверху		уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
10000	0,0018	0,0009	0,0023	0,0032	0,0006	0,0014
20000	0,0037	0,0019	0,0049	0,0068	0,0012	0,0031
30000	0,0062	0,0029	0,0077	0,0107	0,0019	0,0048
40000	0,0082	0,0040	0,0106	0,0146	0,0026	0,0067
50000	0,0106	0,0052	0,0135	0,0187	0,0033	0,0085
60000	0,0131	0,0063	0,0165	0,0228	0,0041	0,0104
70000	0,0150	0,0075	0,0196	0,0270	0,0048	0,0123
80000	0,0176	0,0086	0,0226	0,0313	0,0056	0,0142
90000	0,0198	0,0098	0,0258	0,0356	0,0064	0,0162
100000	0,0224	0,0110	0,0289	0,0399	0,0071	0,0182

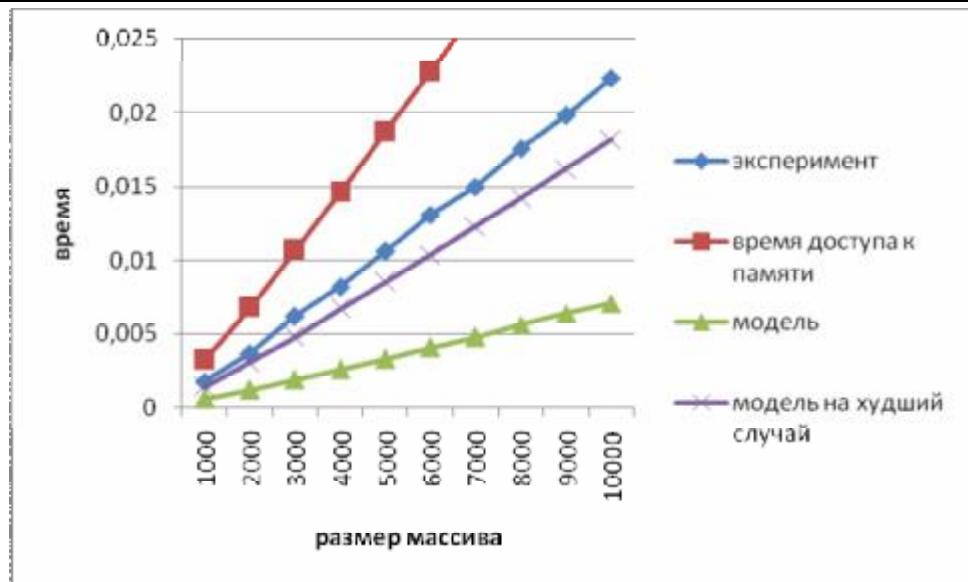


Рис. 9.20. График зависимости экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки от объема исходных данных при использовании четырех потоков

#### 9.4.4. Сортировка с использованием регулярного набора образцов

1. **Организация параллельных вычислений.** Алгоритм сортировки с использованием регулярного набора образцов (*Parallel Sorting by regular sampling*) также является обобщением метода быстрой сортировки (см., например, в [85]).

Упорядочивание данных в соответствии с данным вариантом алгоритма быстрой сортировки осуществляется в ходе выполнения следующих четырех этапов:

- на *первом этапе* сортировки производится упорядочивание имеющихся блоков данных; данная операция может быть выполнена каждым потоком независимо друг от друга при помощи обычного алгоритма быстрой сортировки; далее каждый поток формирует набор из элементов своих блоков с индексами  $0, m, 2m, \dots, (p-1)m$ , где  $m = n/p^2$ ;

- на *втором этапе* выполнения алгоритма все сформированные потоками наборы данных собираются на одном из потоков (*master thread*) системы и сортируются при помощи быстрого алгоритма, таким образом они формируют упорядоченное множество; далее из полученного множества значений из элементов с индексами

$$p + \lfloor p/2 \rfloor - 1, 2p + \lfloor p/2 \rfloor - 1, \dots, (p-1)p + \lfloor p/2 \rfloor$$

формируется новый набор ведущих элементов, который далее используется всеми потоками; в завершение этапа каждый поток выполняет разделение своего блока на  $p$  частей с использованием полученного набора ведущих значений;

- на *третьем этапе* сортировки каждый поток осуществляет «передачу» выделенных ранее частей своего блока всем остальным потокам; «передача» выполняется в соответствии с порядком нумерации - часть  $j$ ,  $0 \leq j < p$ , каждого блока передается потоку с номером  $j$ ;
- на *четвертом этапе* выполнения алгоритма каждый поток выполняет слияние  $p$  полученных частей в один отсортированный блок.

По завершении четвертого этапа исходный набор данных становится отсортированным.

На рис.9.21 приведен пример сортировки массива данных с помощью алгоритма, описанного выше. Следует отметить, что число потоков для данного алгоритма может быть произвольным, в данном примере оно равно 3.

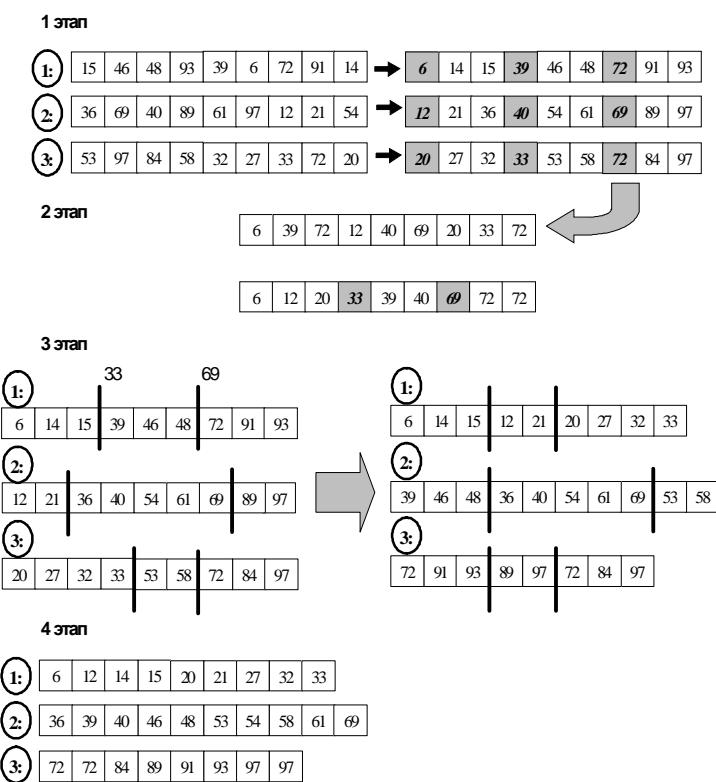


Рис. 9.21. Пример работы алгоритма сортировки с использованием регулярного набора образцов для 3 вычислительных элементов

**2. Анализ эффективности.** Оценим трудоемкость рассмотренного параллельного метода. Пусть, как и ранее,  $n$  есть количество сортируемых данных,  $p$ ,  $p < n$ , обозначает число используемых вычислительных элементов и, соответственно,  $n/p$  есть размер блоков данных, которые обрабатываются параллельными потоками.

В течение *первого этапа* алгоритма каждый поток сортирует свой блок данных с помощью быстрой сортировки, тем самым, длительность выполняемых при этом операций является равной

$$T_p^1 = 1,4 \cdot (n/p) \log_2(n/p) t + 1,4 \cdot \log_2(n/p) \cdot n(a + 64/b), \quad (9.32)$$

где  $t$  есть время выполнения базовой операции сортировки,  $\alpha$  – латентность и  $\beta$  – пропускная способность канала доступа к оперативной памяти.

На *втором этапе* алгоритма один из потоков (*master thread*) собирает наборы из  $p$  элементов со всех остальных процессоров, выполняет сортировку всех полученных данных (общее количество элементов составляет  $p^2$ ), формирует набор из  $p-1$  ведущих элементов. Поскольку в общем случае число вычислительных элементов в системе, а следовательно и число потоков невелико, но массив из  $p^2$  элементов может быть полностью размещен в кэш, и, следовательно, дополнительных затрат на чтение данных из оперативной памяти не требуется. С учетом всех перечисленных действий общая длительность второго этапа составляет

$$T_p^2 = 1,4 \cdot p^2 \cdot \log_2(p^2). \quad (9.33)$$

В ходе выполнения *третьего этапа* алгоритма каждый процессор разделяет свои элементы относительно ведущих элементов на  $p$  частей (поиск очередного места разбиения можно осуществить при помощи алгоритма бинарного поиска)

$$T_p^3 = p \log_2(n/p) \cdot t + n(a + 64/b). \quad (9.34)$$

На *четвертом этапе* алгоритма каждый процессор выполняет слияние  $p$  отсортированных частей в один объединенный блок. Оценка трудоемкости такой операции составляет:

$$T_p^4 = p \cdot \frac{n}{p^2} \cdot t + n \cdot \left( a + \frac{64}{b} \right) = \frac{n}{p} \cdot t + n \cdot \left( a + \frac{64}{b} \right). \quad (9.35)$$

Для выполнения каждого этапа создается параллельная секция; следует учитывать накладные расходы на их организацию и закрытие. С учетом всех полученных соотношений, общее время выполнения алгоритма сортировки с использованием регулярного набора образцов составляет

$$T_p = [(1,4(n/p) + p) \cdot \log_2(n/p) + 1,4p^2 \log_2(p^2) + (n/p)] \cdot t + . \\ + [1,4 \log_2(n/p) + 2] \cdot n(a + 64/b) + 4d \quad (9.36)$$

Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см п. 6.5.4):

$$T_p = [(1,4(n/p) + p) \cdot \log_2(n/p) + 1,4p^2 \log_2(p^2) + (n/p)] \cdot t + . \\ + g[1,4 \log_2(n/p) + 2] \cdot n(a + 64/b) + 4d \quad (9.37)$$

**3. Программная реализация.** Представим возможный вариант параллельной программы быстрой сортировки с использованием регулярного набора образцов. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

Как и ранее, переменные *ThreadNum* и *ThreadId* отвечают за хранение количества потоков в параллельной программе и идентификатора потока соответственно. Эти переменные объявлены как глобальные, для переменной *ThreadId* созданы локальные копии при помощи директивы *threadprivate*. Переменные проинициализированы в функции *InitializeParallelSections*.

Поясним применение дополнительных структур данных. Массив указателей *pDataBlock* хранит указатели на начала блоков в массиве *pData*, которые обрабатываются параллельными потоками. Размеры блоков всех потоков хранятся в массиве *BlockSize*. Таким образом, начало блока, который обрабатывается потоком с номером  $i$ , расположено по адресу *pDataBlock[i]*, и количество элементов в этом блоке равно *BlockSize[i]*.

Массив *LocalSamples* хранит набор из  $p^2$  "локальных" ведущих элементов, выбранных потоками. Поток с номером  $i$  осуществляет запись своих локальных ведущих элементов в ячейки с номерами от  $i \cdot p$  до  $(i+1) \cdot p - 1$ . После сортировки из массива *LocalSamples* выбираются глобальные ведущие элементы и сохраняются в массиве *GlobalSamples*.

Двумерный массив указателей *pDataSubBlock* служит для разделения блоков каждого процесса на подблоки согласно глобальному набору образцов. Указатели на начала подблоков в блоке, за обработку которого отвечает процесс с номером *i*, хранятся в переменных *pDataSubBlock[i][0]*, ..., *pDataSubBlock[i][p-1]*. Размеры подблоков хранятся в двумерном массиве *SubBlockSize*.

Массив *pMergeDataBlock* служит для выполнения операции слияния упорядоченных подблоков.

При выполнении слияния упорядоченных подблоков в каждом потоке используется дополнительный массив *MergeSubBlockSizes*, количество элементов в котором равно количеству параллельных потоков. Перед началом операции слияния в массив заносятся размеры подблоков, предназначенных для слияния на данном потоке. Всякий раз, когда в новый упорядоченный массив добавляется новое значение из подблока какого-либо потока, соответствующий элемент массива *MergeSubBlockSizes* уменьшается на 1. Для поиска наименьшего текущего элемента во всех подблоках, над которыми выполняется операция слияния, используется функция *FindMin*.

```
// Function for parallel quick sorting with regular samples
void ParallelRegularSamplesQuickSort (double* pData, int Size) {
    InitializeParallelSections();

    double** pDataBlock = new double * [ThreadNum];
    int* BlockSize = new int [ThreadNum];
    double* LocalSamples = new double [ThreadNum*ThreadNum];
    double* GlobalSamples = new double [ThreadNum-1];
    double*** pDataSubBlock = new double ** [ThreadNum];
    int** SubBlockSize = new int* [ThreadNum];
    double** pMergeDataBlock = new double* [ThreadNum];

    for (int i=0; i<ThreadNum; i++) {
        BlockSize[i] = Size/ThreadNum;
        pDataBlock[i] = &pData[i*Size/ThreadNum];
        pMergeDataBlock[i] = new double [Size];
        pDataSubBlock[i] = new double* [ThreadNum];
        SubBlockSize[i] = new int [ThreadNum];
    }

    // Local sorting of data blocks
#pragma omp parallel
    {
        SerialQuickSort(pDataBlock[ThreadID], BlockSize[ThreadID]);
    }

    // Samples determination
#pragma omp parallel
    {
        for (int i=0; i<ThreadNum; i++)
            LocalSamples[ThreadID*ThreadNum+i] =
                pDataBlock[ThreadID][i*Size/(ThreadNum*ThreadNum)];
    }

    // Sorting of local samples set
    SerialQuickSort(LocalSamples, ThreadNum*ThreadNum);

    // Global samples determination
    for (int i=1; i<ThreadNum-1; i++)
        GlobalSamples[i-1] = LocalSamples[i*ThreadNum + (ThreadNum/2)-1];
    GlobalSamples[ThreadNum-2] =
        LocalSamples[(ThreadNum-1)*ThreadNum + (ThreadNum/2)];

    // Splitting of data blocks in accordance with global samples
```

```

#pragma omp parallel
{
    pDataSubBlock[ThreadID][0] = pDataBlock[ThreadID];
    int Pos = 0, OldPos = 0;
    for (int i=0; i<ThreadNum-1; i++) {
        OldPos = Pos;
        Pos = BinaryFindPos(pDataBlock[ThreadID], Pos,
            Size/ThreadNum-1, GlobalSamples[i]);
        pDataSubBlock[ThreadID][i+1] = &pDataBlock[ThreadID][Pos];
        SubBlockSize[ThreadID][i] = Pos-OldPos;
    }
    SubBlockSize[ThreadID][ThreadNum-1] = Size/ThreadNum - Pos;
}

// Each thread performs the merging of corresponding subblocks
#pragma omp parallel
{
    int curr = 0;
    double ** pCurr = new double* [ThreadNum];
    int* MergeSubBlockSizes = new int [ThreadNum];
    for (int i=0; i<ThreadNum; i++) {
        pCurr[i] = pDataSubBlock[i][ThreadID];
        MergeSubBlockSizes [i] = SubBlockSize[i][ThreadID];
    }
    while (!IsMergeEnded(MergeSubBlockSizes, ThreadNum)) {
        int MinPos = FindMin(pCurr, ThreadNum, MergeSubBlockSizes);
        pMergeDataBlock[ThreadID][curr] = *(pCurr[MinPos]);
        MergeSubBlockSizes[MinPos]--;
        if (MergeSubBlockSizes[MinPos] != 0)
            pCurr[MinPos]++;
        curr++;
    } // while
    BlockSize[ThreadID] = curr;
    delete [] pCurr;
    delete [] MergeSubBlockSizes;
} // pragma omp parallel

// Copying the data from the pMergeDataBlock arrays to the initial array
int NewCurr = 0;
for (int i=0; i<ThreadNum; i++)
    for (int j=0; j<BlockSize[i]; j++)
        pData[NewCurr++] = pMergeDataBlock[i][j];

for (int i=0; i<ThreadNum; i++) {
    delete [] pDataSubBlock [i];
    delete [] pMergeDataBlock[i];
    delete [] SubBlockSize[i];
}
delete [] pDataBlock;
delete [] pDataSubBlock;
delete [] pMergeDataBlock;
delete [] SubBlockSize;
delete [] BlockSize;
delete [] LocalSamples;
delete [] GlobalSamples;
}

```

Для разделения блоков потоков на подблоки согласно глобальному регулярному набору образцов используется функция *BinaryFindPos*, которая осуществляет бинарный поиск заданного элемента *Elem* в упорядоченном массиве *Array*, начиная с элемента с индексом *first* и заканчивая элементом с индексом *last*. В качестве возвращаемого значения выступает номер позиции, на которой должен быть расположен искомый элемент с тем, чтобы массив *Array* остался упорядоченным.

```

// Function for binary searching
int BinaryFindPos (double* Array, int first, int last, double Elem) {
    if (Elem<Array[first]) return first;
    if (Elem>Array[last]) return (last);
    int middle;
    while (last-first > 1) {
        middle = (first+last)/2;
        if (Array[middle] == Elem)
            return middle;
        if (Array[middle]>ELEM) last = middle;
        if (Array[middle]<ELEM) first = middle;
    }
    return last;
}

```

**4. Результаты вычислительных экспериментов.** Вычислительные эксперименты для оценки эффективности параллельного варианта метода быстрой сортировки с использованием регулярного набора образцов проводились при условиях, указанных в п. 9.2.1. Вычисления проводились на двух процессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Результаты вычислительных экспериментов приведены в таблице 9.19. Времена выполнения алгоритмов указаны в секундах.

**Таблица 9.19.** Результаты вычислительных экспериментов для параллельного метода быстрой сортировки с использованием регулярного набора образцов

Размер массива	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
10000	0,0029	0,0043	0,6695	0,0012	2,3282
20000	0,0062	0,0057	1,0751	0,0024	2,5599
30000	0,0100	0,0073	1,3741	0,0038	2,6240
40000	0,0133	0,0112	1,1810	0,0054	2,4552
50000	0,0172	0,0126	1,3647	0,0062	2,7520
60000	0,0209	0,0154	1,3611	0,0078	2,6695
70000	0,0245	0,0154	1,5861	0,0088	2,7884
80000	0,0278	0,0181	1,5386	0,0131	2,1299
90000	0,0318	0,0203	1,5650	0,0117	2,7088
100000	0,0366	0,0229	1,5952	0,0132	2,7711

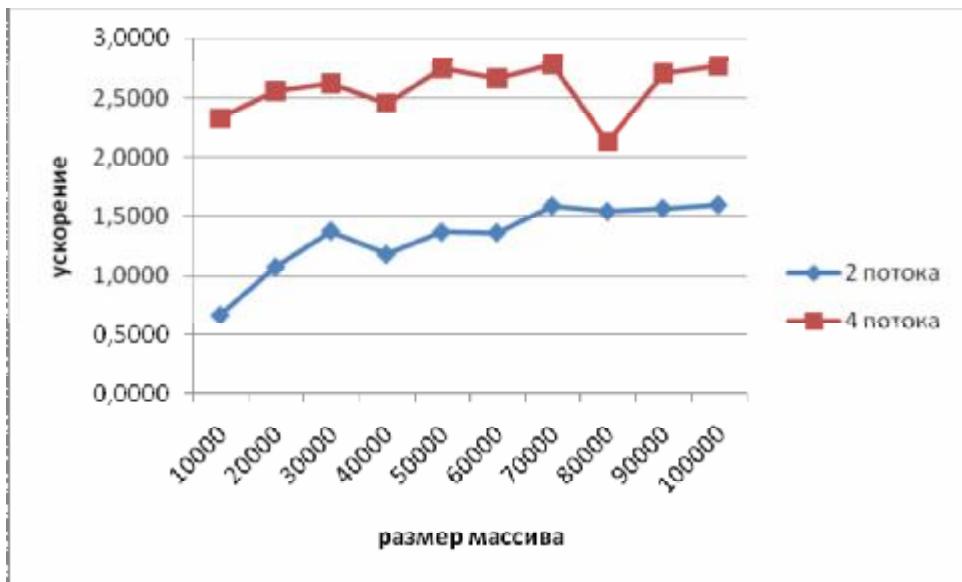


Рис. 9.22. Зависимость ускорения от количества исходных данных при выполнении параллельного метода быстрой сортировки с использованием регулярного набора образцов

В таблицах 9.20 и 9.21 и на рис. 9.23 и 9.24 представлены результаты сравнения времени выполнения  $T_p$  параллельного метода обобщенной быстрой сортировки с использованием двух и четырех потоков со временем  $T_p^*$ , полученным при помощи модели (9.37). Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0785, а для четырех потоков значение этой величины было оценено как 0,1963. Как и ранее, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

**Таблица 9.20.** Сравнение экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки с использованием регулярного набора образцов с использованием двух потоков

Размер матриц	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 9. 36 – оценка сверху		Модель 9.37 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
10000	0,0043	0,0018	0,0025	0,0043	0,0002	0,0020
20000	0,0057	0,0039	0,0054	0,0093	0,0004	0,0043
30000	0,0073	0,0061	0,0084	0,0145	0,0007	0,0068
40000	0,0112	0,0084	0,0115	0,0199	0,0009	0,0093
50000	0,0126	0,0107	0,0147	0,0254	0,0012	0,0119
60000	0,0154	0,0131	0,0179	0,0310	0,0014	0,0145
70000	0,0154	0,0155	0,0212	0,0367	0,0017	0,0171
80000	0,0181	0,0179	0,0245	0,0424	0,0019	0,0198
90000	0,0203	0,0204	0,0279	0,0482	0,0022	0,0225
100000	0,0229	0,0228	0,0313	0,0541	0,0025	0,0253

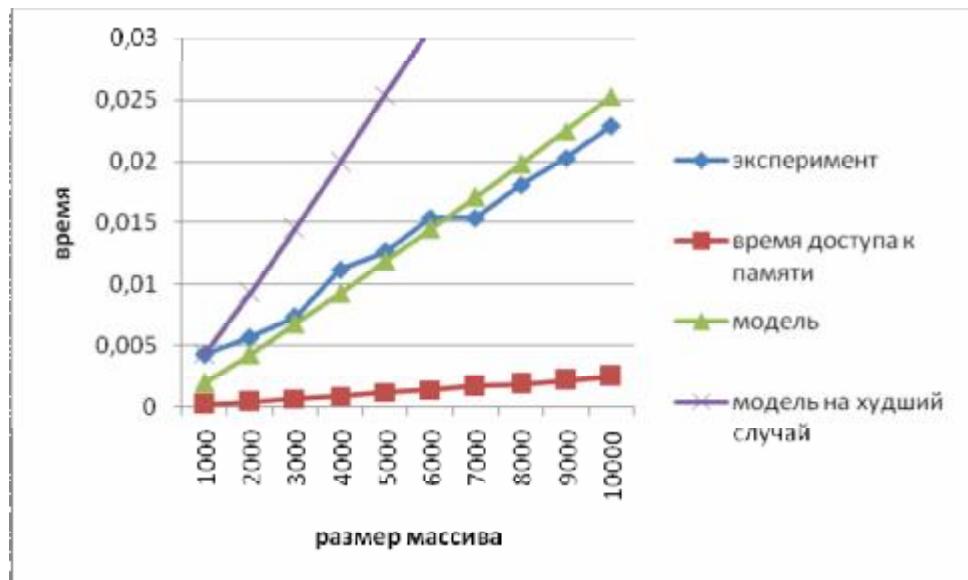


Рис. 9.23. График зависимости экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки с использованием регулярного набора образцов от объема исходных данных при использовании двух потоков

**Таблица 9.21.** Сравнение экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки с использованием регулярного набора образцов с использованием четырех потоков

Размер матриц	$T_p$	$T_p^* (calc)$ (модель)	Модель 9. 36 – оценка сверху		Модель 9.37 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
10000	0,0012	0,0008	0,0023	0,0032	0,0005	0,0013
20000	0,0024	0,0018	0,0050	0,0069	0,0010	0,0028
30000	0,0038	0,0029	0,0079	0,0107	0,0015	0,0044
40000	0,0054	0,0039	0,0108	0,0147	0,0021	0,0060
50000	0,0062	0,0050	0,0138	0,0188	0,0027	0,0077
60000	0,0078	0,0061	0,0168	0,0230	0,0033	0,0094
70000	0,0088	0,0073	0,0199	0,0272	0,0039	0,0112
80000	0,0101	0,0084	0,0231	0,0315	0,0045	0,0129
90000	0,0117	0,0096	0,0262	0,0358	0,0051	0,0147
100000	0,0132	0,0107	0,0294	0,0401	0,0058	0,0165

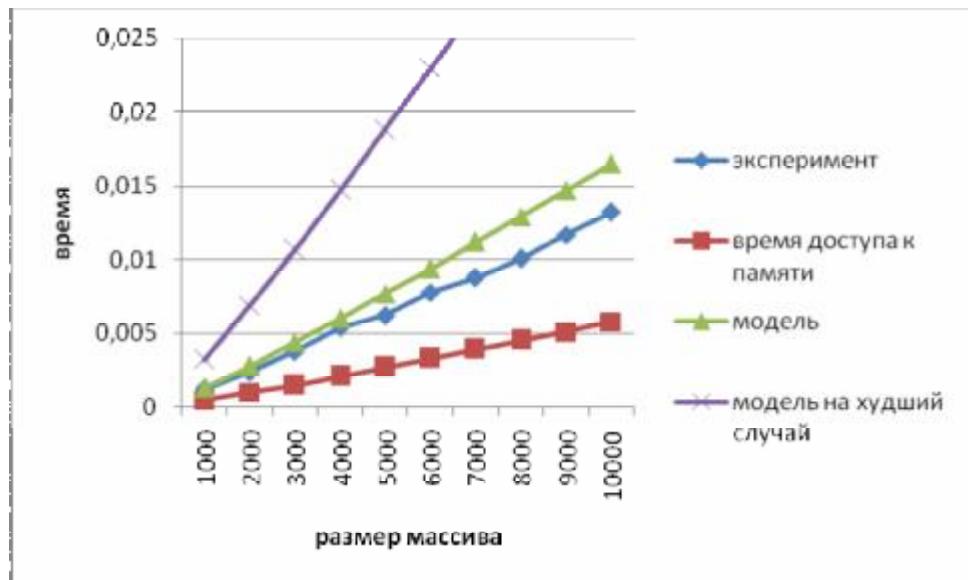


Рис. 9.24. График зависимости экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки с использованием регулярного набора образцов от объема исходных данных при использовании четырех потоков

## 9.5. Краткий обзор главы

В главе рассматривается часто встречающаяся в приложениях задача упорядочения данных, для решения которой в рамках данного учебного материала выбраны широко известные алгоритмы пузырьковой сортировки, сортировки Шелла и быстрой сортировки. При изложении методов сортировки основное внимание уделяется возможным способам распараллеливания алгоритмов, анализу эффективности и сравнению получаемых теоретических оценок с результатами выполненных вычислительных экспериментов.

*Алгоритм пузырьковой сортировки* в исходном виде практически не поддается распараллеливанию в силу последовательного выполнения основных итераций метода. Для введения необходимого параллелизма рассматривается обобщенный вариант алгоритма - *метод чет-нечетной перестановки*. Суть обобщения состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода в зависимости от четности номера итерации сортировки. Сравнения пар значений упорядочиваемого набора данных на итерациях метода чет-нечетной перестановки являются независимыми и могут быть выполнены параллельно.

Для *алгоритма Шелла* рассматривается схема распараллеливания при представлении множества потоков параллельной программы в виде гиперкуба. При таком представлении оказывается возможным организация взаимодействия потоков и выполнить операции сравнения и разделения подблоков, расположенных далеко друг от друга при линейной нумерации. Как правило, такая организация вычислений позволяет уменьшить количество выполняемых итераций алгоритма сортировки.

Для *алгоритма быстрой сортировки* приводятся три схемы распараллеливания. Первые две схемы также основаны на представлении множества потоков параллельной программы в виде гиперкуба. Основная итерация вычислений состоит в выборе одним из потоков ведущего элемента. После получения ведущего элемента потоки проводят разделение своих блоков, и получаемые части блоков передаются между попарно связанными потоками. В результате выполнения подобной итерации исходный гиперкуб оказывается разделенным на 2 гиперкуба меньшей размерности, к которым, в свою очередь, может быть применена приведенная выше схема вычислений.

При применении алгоритма быстрой сортировки одним из основных моментов является правильность выбора ведущего элемента. Оптимальная ситуация состоит в выборе такого значения ведущего элемента, при котором блоки данных разделяются на части одинакового размера. В общем случае, при произвольно сгенерированных исходных данных достижение такой ситуации является достаточно сложной задачей. В первой схеме предлагается выбирать ведущий элемент, например, на основании информации об максимальном и минимальном значениях в сортируемом наборе. Во второй схеме блоки данных предварительно упорядочиваются с тем, чтобы взять средний элемент блока как ведущее значение.

Третья схема распараллеливания алгоритма быстрой сортировки основывается на многоуровневой схеме формирования множества ведущих элементов. Такой подход может быть применен для произвольного количества потоков и приводит, как правило, к лучшей балансировке распределения данных между процессорами.

Сравнение показателей ускорения различных параллельных алгоритмов сортировки в зависимости от объема исходных данных представлено на рис. 9.25.

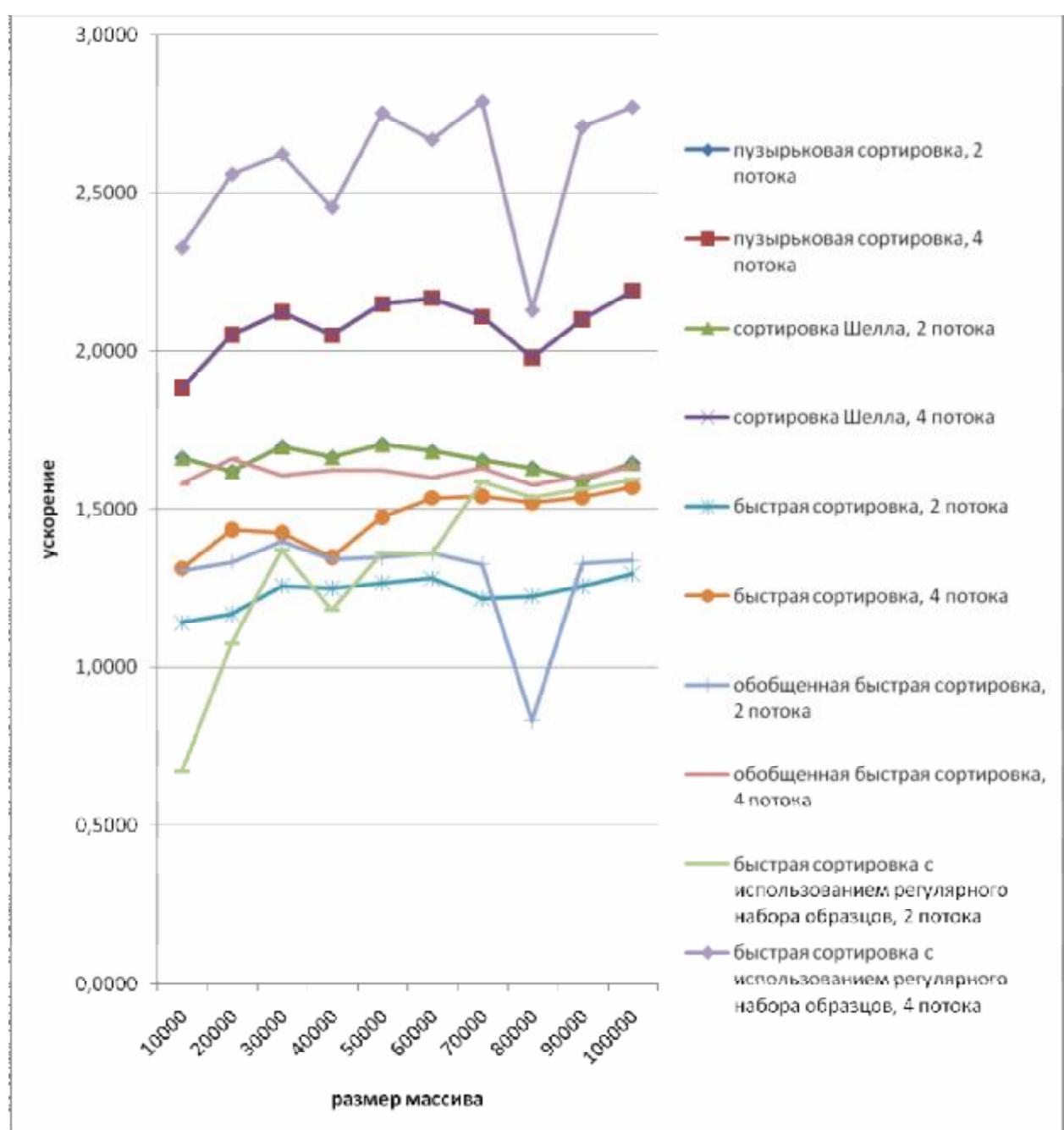


Рис. 9.25. Ускорение параллельных алгоритмов сортировки данных

## 9.6. Обзор литературы

Возможные способы решения задачи упорядочения данных широко обсуждаются в литературе; один из наиболее полных обзоров *алгоритмов сортировки* содержится в работе [71], может быть рекомендована также работа [17].

Параллельные варианты *алгоритма пузырьковой сортировки* и *сортировки Шелла* рассматриваются в [72].

Схемы распараллеливания *быстрой сортировки* при представлении топологии сети передачи данных в виде гиперкуба описаны в [72,85]. *Сортировка с использованием регулярного набора образцов (parallel sorting by regular sampling)* представлена в работе [85].

Полезной при рассмотрении вопросов параллельных вычислений для сортировки данных может оказаться работа [37].

## 9.7. Контрольные вопросы

1. В чем состоит постановка задачи сортировки данных?
2. Приведите несколько примеров алгоритмов сортировки? Какова вычислительная сложность приведенных алгоритмов?
3. Какая операция является базовой для задачи сортировки данных?
4. В чем суть параллельного обобщения базовой операции задачи сортировки данных?
5. Что представляет собой алгоритм чет-нечетной перестановки?
6. В чем состоит параллельный вариант алгоритма Шелла? Какие основные отличия этого варианта параллельного алгоритма сортировки от метода чет-нечетной перестановки?
7. Что представляет собой параллельный вариант алгоритма быстрой сортировки?
8. Что зависит от правильного выбора ведущего элемента для параллельного алгоритма быстрой сортировки?
9. Какие способы выбора ведущего элемента могут быть предложены?
10. В чем состоит алгоритм сортировки с использованием регулярного набора образцов?

## 9.8. Задачи и упражнения

1. Выполните реализацию параллельного алгоритма пузырьковой сортировки. Проведите эксперименты. Постройте теоретические оценки. Сравните получаемые теоретические оценки с результатами экспериментов.
2. Выполните реализацию параллельного алгоритма быстрой сортировки по одной из приведенных схем. Определите значения параметров латентности, пропускной способности и времени выполнения базовой операции для используемой вычислительной системы и получите оценки показателей ускорения и эффективности для реализованного метода параллельных вычислений.
3. Разработайте параллельную схему вычислений для широко известного алгоритма сортировки слиянием (подробное описание метода может быть получено, например, в работах [17,71]). Выполните реализацию разработанного алгоритма и постройте все необходимые теоретические оценки сложности метода.

Глава 10. Обработка графов .....	1
10.1. Задача поиска всех кратчайших путей .....	3
10.1.1. Последовательный алгоритм Флойда.....	3
10.1.2. Разделение вычислений на независимые части .....	4
10.1.3. Масштабирование и распределение подзадач по процессорам.....	4
10.1.4. Анализ эффективности параллельных вычислений .....	4
10.1.5. Программная реализация .....	5
10.1.6. Результаты вычислительных экспериментов .....	7
10.2. Задача нахождения минимального охватывающего дерева .....	10
10.2.1. Последовательный алгоритм Прима .....	11
10.2.2. Программная реализация последовательного алгоритма Прима .....	11
10.2.3. Разделение вычислений на независимые части.....	14
10.2.4. Анализ эффективности параллельных вычислений.....	14
10.2.5. Программная реализация параллельного алгоритма Прима.....	15
10.2.6. Результаты вычислительных экспериментов .....	17
10.3. Задача оптимального разделения графов.....	20
10.3.1. Постановка задачи оптимального разделения графов .....	21
10.3.2. Метод рекурсивного деления пополам.....	22
10.3.3. Геометрические методы.....	22
10.3.4. Комбинаторные методы.....	24
10.3.5. Сравнение алгоритмов разбиения графов.....	26
10.4. Краткий обзор главы .....	27
10.5. Обзор литературы .....	28
10.6. Контрольные вопросы .....	28
10.7. Задачи и упражнения.....	28

## Глава 10. Обработка графов

Математические модели в виде *графов* широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор *типовых алгоритмов обработки графов*. Рассмотрению вопросов теории графов, алгоритмов моделирования, анализу и решению задач на графах посвящено достаточно много различных изданий, в качестве возможного руководства по данной тематике может быть рекомендована работа [17].

Пусть  $G$  есть график

$$G = (V, R),$$

для которого набор вершин  $v_i$ ,  $1 \leq i \leq n$ , задается множеством  $V$ , а список дуг графа

$$r_j = (v_{s_j}, v_{t_j}), \quad 1 \leq j \leq m,$$

определяется множеством  $R$ . В общем случае дугам графа могут приписываться некоторые числовые характеристики (*веса*)  $w_j$ ,  $1 \leq j \leq m$  (*взвешенный график*). Пример взвешенного графа приведен на рис. 10.1.

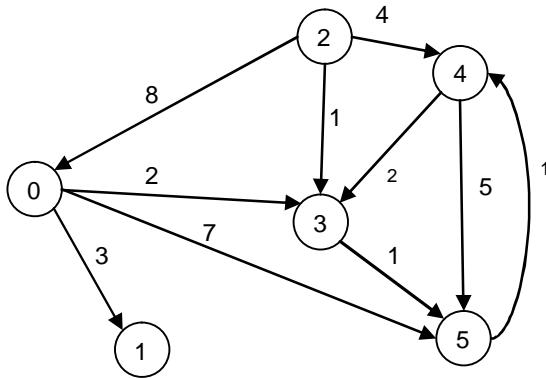


Рис. 10.1. Пример взвешенного ориентированного графа

Известны различные способы задания графов. При малом количестве дуг в графе (т.е.  $m << n^2$ ) целесообразно использовать для определения графов списки, перечисляющие имеющиеся в графах дуги. Такие списки получили название *списков примыканий*. Список примыканий графа  $G=(V, R)$  с числом вершин  $n$  записывается в виде одномерного массива длины  $n$ , каждый элемент которого представляет собой ссылку на список. Такой список приписан каждой вершине графа, и он содержит по одному элементу на каждую вершину графа, соседнюю с данной. Каждое звено списка хранит номер соседней вершины и вес ребра. Так, например, список примыканий, соответствующий графу, изображеному на рис. 10.1, приведен на рис. 10.2.

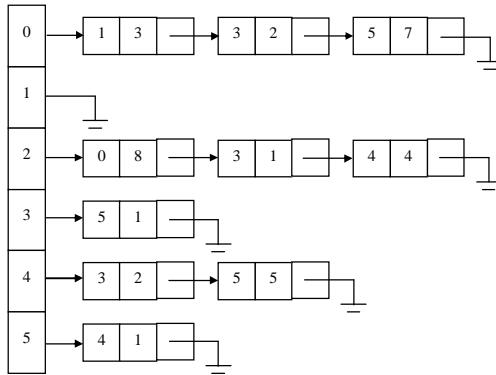


Рис. 10.2. Список примыканий для графа из рис. 10.1

Представление достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т.е.  $m \sim n^2$ ), может быть эффективно обеспечено при помощи *матрицы смежности*

$$A = (a_{ij}), \quad 1 \leq i, j \leq n,$$

ненулевые значения элементов которой соответствуют дугам графа

$$a_{ij} = \begin{cases} w(v_i, v_j), & \text{если } (v_i, v_j) \in R, \\ 0, & \text{если } i = j, \\ \infty, & \text{иначе.} \end{cases}$$

(для обозначения отсутствия ребра между вершинами в матрице смежности на соответствующей позиции используется знак бесконечности, при вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число). Так, например, матрица смежности, соответствующая графу на рис. 10.1, приведена на рис. 10.3.

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Рис. 10.3. Матрица смежности для графа из рис. 10.1

Как положительный момент такого способа представления графов можно отметить, что использование матрицы смежности позволяет применять при реализации вычислительных процедур анализа графов матричные алгоритмы обработки данных.

Далее мы рассмотрим способы параллельной реализации алгоритмов на графах на примере задачи *поиска кратчайших путей* между всеми парами пунктов назначения и задачи *выделения минимального охватывающего дерева (остова)* графа. Кроме того, мы рассмотрим задачу *оптимального разделения графов*, широко используемую для организации параллельных вычислений. Для представления графов, подлежащих обработке, при рассмотрении всех перечисленных задач будут использоваться матрицы смежности. Для представления минимального охватывающего дерева, полученного в результате выполнения алгоритма Прима, будет использоваться список примыканий.

## 10.1. Задача поиска всех кратчайших путей

Исходной информацией для задачи является взвешенный граф  $G = (V, R)$ , содержащий  $n$  вершин ( $|V| = n$ ), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать *ориентированным*, т.е., если из вершины  $i$  есть ребро в вершину  $j$ , то из этого не следует наличие ребра из  $j$  в  $i$ . В случае, если вершины все же соединены взаимообратными ребрами, то веса, приписанные им, могут не совпадать. Рассмотрим задачу, в которой для имеющегося графа  $G$  требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется *алгоритм Флойда (Floyd)* (см, например, [17]).

### 10.1.1. Последовательный алгоритм Флойда

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок  $n^3$ . В общем виде данный алгоритм может быть представлен следующим образом:

```
// Алгоритм 10.1
// Serial Floyd algorithm
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            A[i, j] = min(A[i, j], A[i, k]+A[k, j]);
```

Алгоритм 10.1. Общая схема алгоритма Флойда

(реализация операции выбора минимального значения *min* должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе

выполнения алгоритма матрица смежности  $A$  изменяется, после завершения вычислений в матрице  $A$  будет храниться требуемый результат - длины минимальных путей для каждой пары вершин исходного графа.

Дополнительная информация и доказательство правильности алгоритма Флойда могут быть получены, например, в работе [17].

### 10.1.2. Разделение вычислений на независимые части

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимального значения (см. Алгоритм 10.1). Данная операция является достаточно простой и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы  $A$ .

Покажем корректность такого способа организации параллелизма. Для этого нужно доказать, что операции обновления значений матрицы  $A$  на одной и той же итерации внешнего цикла  $k$  могут выполняться независимо. Иными словами, следует показать, что на итерации  $k$  не происходит изменения элементов  $A_{ik}$  и  $A_{kj}$  ни для одной пары индексов  $(i, j)$ . Рассмотрим выражение, по которому происходит изменение элементов матрицы  $A$ :

$$A_{ij} \leftarrow \min(A_{ij}, A_{ik} + A_{kj}).$$

Для  $i=k$  получим

$$A_{kj} \leftarrow \min(A_{kj}, A_{kk} + A_{kj}),$$

но тогда значение  $A_{kj}$  не изменится, т.к.  $A_{kk}=0$ .

Для  $j=k$  выражение преобразуется к виду

$$A_{ik} \leftarrow \min(A_{ik}, A_{ik} + A_{kk}),$$

что также показывает неизменность значений  $A_{ik}$ . Как результат, необходимые условия для организации параллельных вычислений обеспечены и, тем самым, в качестве *базовой подзадачи* может быть использована операция обновления элементов матрицы  $A$  (для указания подзадач будем использовать индексы обновляемых в подзадачах элементов).

### 10.1.3. Масштабирование и распределение подзадач по процессорам

Как правило, число доступных вычислительных элементов  $p$  существенно меньше, чем число базовых задач  $n^2$  ( $p \ll n^2$ ). Возможный способ укрупнения вычислений состоит в использовании *ленточной схемы* разбиения матрицы  $A$  – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк (*горизонтальное разбиение*) или столбцов (*вертикальное разбиение*) матрицы  $A$ . Эти два типа разбиения практически равноправны – учитывая дополнительный момент, что для алгоритмического языка С массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы  $A$  на горизонтальные полосы.

### 10.1.4. Анализ эффективности параллельных вычислений

Как и ранее, при анализе эффективности параллельного алгоритма Флойда будем предполагать, что время выполнения складывается из времени вычислений (которые могут быть выполнены вычислительными элементами параллельно) и времени, необходимого на загрузку необходимых данных из оперативной памяти в кэш. Доступ к памяти осуществляется строго последовательно.

Для выполнения алгоритма Флойда над графом, содержащим  $n$  вершин, требуется выполнение  $n$  итераций алгоритма, на каждой из которых параллельно выполняется обновление всех элементов матрицы смежности. Значит, время выполнения вычислений составляет:

$$T_p(\text{calc}) = n \cdot \frac{n^2}{p} \cdot t, \quad (10.1)$$

где  $t$  есть время выполнения операции выбора минимального значения.

Если количество вершин в графе настолько велико, что описывающая граф матрица смежности не может быть полностью помещена в кэш, то на каждой итерации внешнего цикла  $k$  выполняется вытеснение «первых» элементов матрицы для того, чтобы записать на их место значения, располагаемые в конце матрицы. Для перехода к выполнению следующей итерации необходимо снова считать значения из начала матрицы. Таким образом, происходит повторное считывание всех элементов матрицы из оперативной памяти в кэш, и затраты на доступ к памяти составляют:

$$T_p(\text{mem}) = n \cdot \frac{64n^2}{b}, \quad (10.2)$$

где  $\beta$  есть пропускная способность канала доступа к оперативной памяти

Если, как и в предыдущих главах, учесть латентность памяти, то время доступа к памяти можно получить по следующей формуле:

$$T_p(\text{mem}) = n^3 \cdot \left( a + \frac{64}{b} \right), \quad (10.3)$$

При получении итоговой оценки времени выполнения параллельного алгоритма Флойда необходимо также учитывать затраты на организацию и закрытие параллельных секций:

$$T_p = \frac{n^3}{p} \cdot t + n^3 \cdot \left( a + \frac{64}{b} \right) + nd, \quad (10.4)$$

где  $d$  есть накладные расходы на организацию параллельности на каждой итерации алгоритма.

Построенная модель является моделью на худший случай. Для более точной оценки необходимо учесть частоту кэш промахов  $\gamma$  (см. п. 6.5.4):

$$T_p = \frac{n^3}{p} \cdot t + g \cdot n^3 \cdot \left( a + \frac{64}{b} \right) + nd, \quad (10.5)$$

### 10.1.5. Программная реализация

Представим возможный вариант программы, выполняющей параллельный алгоритм Флойда поиска всех кратчайших путей.

**1. Главная функция программы.** Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 10.1
// Parallel Floyd algorithm
void main(int argc, char* argv[]) {
    double *pMatrix; // Adjacency matrix
    int Size; // Size of adjacency matrix

    // Data initialization
    ProcessInitialization(pMatrix, Size);

    // Parallel Floyd algorithm
    ParallelFloyd(pMatrix, Size);
```

```

    // Process termination
    ProcessTermination(pMatrix);
}

```

**2. Функция ProcessInitialization.** Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества вершин в графе, выделения памяти для хранения матрицы смежности и для заполнения этой матрицы значениями. Начальные значения элементов матрицы смежности задаются в функции *RandomDataInitialization*.

```

// Function for memory allocation and data initialization
void ProcessInitialization(double *pMatrix, int& Size) {
    do {
        printf("Enter the number of vertices: ");
        scanf("%d", &Size);
        if(Size <= 2)
            printf("The number of vertices should be greater than two\n");
    } while(Size <= 2);

    printf("Using graph with %d vertices\n", Size);

    // Allocate memory for the adjacency matrix
    pMatrix = new double[Size * Size];

    // Data initialization
    RandomDataInitialization(pMatrix, Size);
}

```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

**3. Функция ParallelFloyd.** Данная функция выполняет параллельный алгоритм Флойда поиска кратчайших путей для всех пар вершин.

```

// Function for parallel Floyd algorithm's execution
void ParallelFloyd(double *pMatrix, int Size) {
    double t1, t2;
    for(int k = 0; k < Size; k++)
#pragma omp parallel for private (t1, t2)
        for(int i = 0; i < Size; i++)
            for(int j = 0; j < Size; j++)
                if((pMatrix[i * Size + k] != -1) &&
                   (pMatrix[k * Size + j] != -1)) {
                    t1 = pMatrix[i * Size + j];
                    t2 = pMatrix[i * Size + k] + pMatrix[k * Size + j];
                    pMatrix[i * Size + j] = Min(t1, t2);
                }
}

```

**4. Функция Min.** Функция *Min* вычисляет наименьшее из двух чисел, учитывая используемый метод обозначения несуществующих дуг в матрице смежности (в рассматриваемой реализации используется значение -1).

```

double Min(double A, double B) {
    double Result = (A < B) ? A : B;

    if((A < 0) && (B >= 0)) Result = B;
    if((B < 0) && (A >= 0)) Result = A;
    if((A < 0) && (B < 0)) Result = -1;

    return Result;
}

```

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

#### 10.1.6. Результаты вычислительных экспериментов

Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Для снижения сложности построения теоретических оценок времени выполнения алгоритмов при компиляции и построении программ для проведения вычислительных экспериментов функция оптимизации кода компилятором была отключена (результаты оценки влияния компиляторной оптимизации на эффективность программного кода приведены в п. 7.2.4).

Результаты вычислительных экспериментов приведены в таблице 10.1. Времена выполнения алгоритмов указаны в секундах.

**Таблица 10.1.** Результаты вычислительных экспериментов для параллельного алгоритма Флойда  
(при использовании двух и четырех вычислительных ядер)

количество вершин в графе	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
100	0,0549	0,0259	2,1227	0,0133	4,1305
200	0,4389	0,2004	2,1898	0,1024	4,2870
300	1,4739	0,6708	2,1971	0,3416	4,3151
400	3,4846	1,5809	2,2042	0,8069	4,3187
500	6,8002	3,0822	2,2063	1,5710	4,3285
600	11,7399	5,3205	2,2065	2,7112	4,3301
700	18,6608	8,4392	2,2112	4,2986	4,3411
800	27,9942	12,5910	2,2234	6,3962	4,3767
900	39,9018	17,9152	2,2273	9,0951	4,3872
1000	54,7534	24,5916	2,2265	12,4759	4,3887

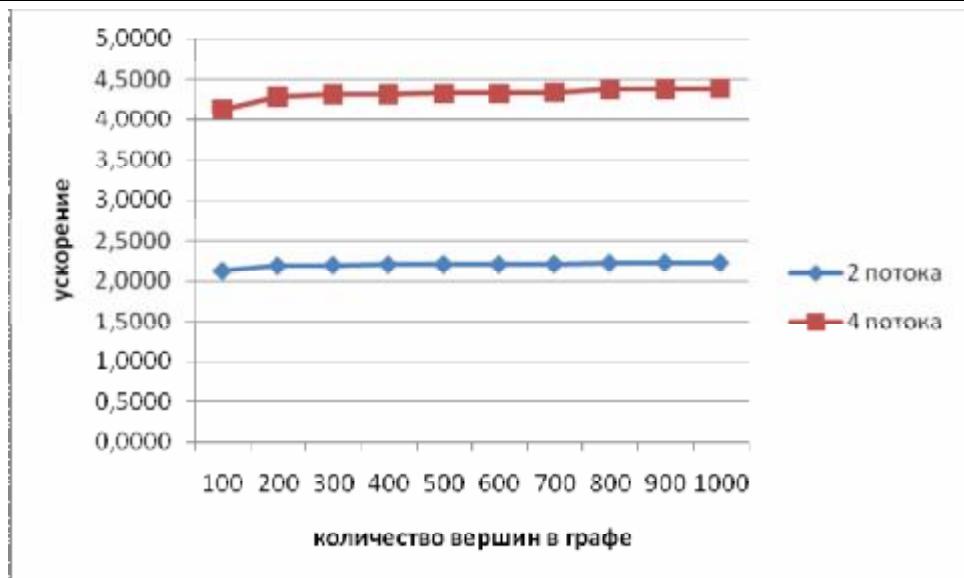


Рис. 10.4. Зависимость ускорения от количества исходных данных при выполнении параллельного алгоритма Флойда

В таблице 10.2 и 10.3 и на рис. 10.4 и 10.5 представлены результаты сравнения времени выполнения параллельного алгоритма Флойда с использованием двух потоков со временем, полученным при помощи моделей (10.4) и (10.5).

Для того, чтобы оценить время одной операции выбора минимального значения  $\tau$ , измерим время выполнения последовательного алгоритма Флойда при малых объемах данных, таких, чтобы матрица смежности, описывающая граф, могла быть полностью помещена в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицу случайными значениями. Выполнение этого действия гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение  $\tau$ , равное 55,178 нс.

Оценки времени латентности  $\alpha$  и величины пропускной способности канала доступа к оперативной памяти  $\beta$  проводилась в п. 6.5.4 и определены для используемого вычислительного узла как  $\alpha = 8,31$  нс. и  $\beta = 12,44$  Гб/с. Величина накладных расходов  $\delta$  на параллельность была оценена в главе 6 и составляет 0,25·мкс. Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,0083, а для четырех потоков значение этой величины была оценена как 0,0090.

Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах сортируемого массива составляет не более 1%.

**Таблица 10.2.** Сравнение экспериментального и теоретического времени выполнения параллельного метода Флойда с использованием двух потоков

Размер массива	$T_p$	$T_p^* (calc)$ (модель)	Модель 10.4 – оценка сверху		Модель 10.5 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
100	0,0259	0,0276	0,0131	0,0407	0,0001	0,0277
200	0,2004	0,2208	0,1048	0,3256	0,0009	0,2216
300	0,6708	0,7450	0,3537	1,0987	0,0029	0,7479
400	1,5809	1,7658	0,8385	2,6043	0,0070	1,7728
500	3,0822	3,4488	1,6377	5,0864	0,0136	3,4623
600	5,3205	5,9594	2,8299	8,7893	0,0235	5,9829
700	8,4392	9,4632	4,4938	13,9570	0,0373	9,5005
800	12,5910	14,1258	6,7079	20,8337	0,0557	14,1814
900	17,9152	20,1126	9,5509	29,6635	0,0793	20,1919
1000	24,5916	27,5893	13,1014	40,6906	0,1087	27,6980



Рис. 10.5. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Флойда от объема исходных данных при использовании двух потоков

**Таблица 10.3.** Сравнение экспериментального и теоретического времени выполнения параллельного метода Флойда с использованием четырех потоков

Размер массива	$T_p$	$T_p^* (calc)$ (модель)	Модель 10.4 – оценка сверху		Модель 10.5 – уточненная оценка	
			$T_p^* (mem)$	$T_p^*$	$T_p^* (mem)$	$T_p^*$
100	0,0133	0,0138	0,0131	0,0269	0,0001	0,0139
200	0,1024	0,1104	0,1048	0,2152	0,0009	0,1113
300	0,3416	0,3725	0,3537	0,7263	0,0032	0,3757
400	0,8069	0,8829	0,8385	1,7214	0,0075	0,8905
500	1,5710	1,7244	1,6377	3,3621	0,0147	1,7392
600	2,7112	2,9798	2,8299	5,8097	0,0255	3,0052
700	4,2986	4,7317	4,4938	9,2255	0,0404	4,7721
800	6,3962	7,0630	6,7079	13,7709	0,0604	7,1234
900	9,0951	10,0564	9,5509	19,6073	0,0860	10,1424
1000	12,4759	13,7948	13,1014	26,8961	0,1179	13,9127



Рис. 10.6. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Флойда от объема исходных данных при использовании четырех потоков

## 10.2. Задача нахождения минимального охватывающего дерева

*Охватывающим деревом* (или *остовом*) неориентированного графа  $G$  называется подграф  $T$  графа  $G$ , который является деревом и содержит все вершины из  $G$ . Определив вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, под *минимально охватывающим деревом (МОД)*  $T$  будем понимать охватывающее дерево минимального веса. Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием соединительных линий связи минимальной длины. Пример взвешенного неориентированного графа и соответствующего ему минимального охватывающего дерева приведен на рис. 10.7.

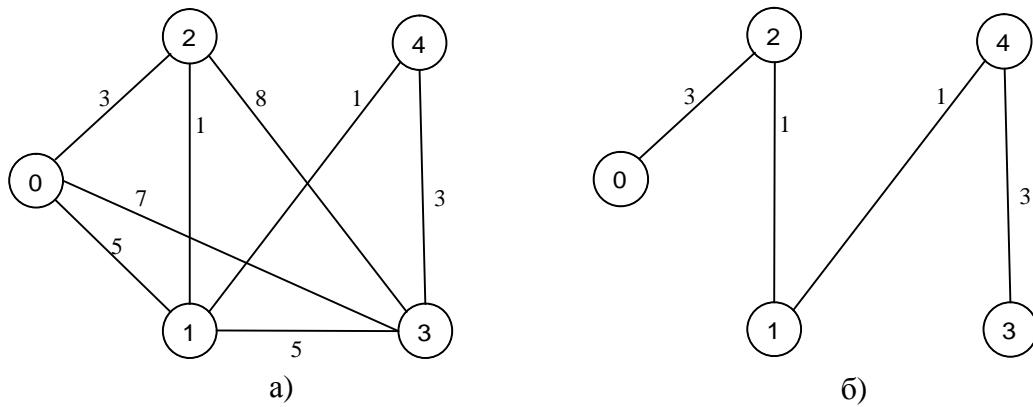


Рис. 10.7. Пример (а) взвешенного неориентированного графа и соответствующему ему (б) минимально охватывающему дерева

Дадим общее описание алгоритма решения поставленной задачи, известного под названием *метода Прима (Prim)*, более полная информация может быть получена, например, в [17].

### 10.2.1. Последовательный алгоритм Прима

Алгоритм начинает работу с произвольной вершиной графа, выбираемой в качестве корня дерева, и в ходе последовательно выполняемых итераций расширяет конструируемое дерево до МОД. Пусть  $V_T$  есть множество вершин, уже включенных алгоритмом в МОД, а величины  $d_i$ ,  $1 \leq i \leq n$ , характеризуют дуги минимальной длины от вершин, еще не включенных в дерево, до множества  $V_T$ , т.е.

$$\forall i \notin V_T \Rightarrow d_i = \min \{w(i, u) : u \in V_T, (i, u) \in R\}$$

(если для какой-либо вершины  $i \notin V_T$  не существует ни одной дуги в  $V_T$ , значение  $d_i$  устанавливается в  $\infty$ ). В начале работы алгоритма выбирается корневая вершина МОД с и полагается  $V_T = \{s\}$ ,  $d_s = 0$ .

Действия, выполняемые на каждой итерации алгоритма Прима, состоят в следующем:

- определяются значения величин  $d_i$  для всех вершин, еще не включенных в состав МОД;
- выбирается вершина  $t$  графа  $G$ , имеющая дугу минимального веса до множества

$$V_T \quad t : d_t = \min(d_i), \quad i \notin V_T ;$$

- вершина  $t$  включается в  $V_T$ .

После выполнения  $n - 1$  итерации метода МОД будет сформировано. Вес этого дерева может быть получен при помощи выражения

$$W_T = \sum_{i=1}^n d_i .$$

Трудоемкость нахождения МОД характеризуется квадратичной зависимостью от числа вершин графа  $O(n^2)$ .

### 10.2.2. Программная реализация последовательного алгоритма Прима

Представим возможный вариант программы, выполняющей последовательный алгоритм Прима построения минимального охватывающего дерева.

**1. Главная функция программы.** Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 10.2
// Serial Prim algorithm
void main(int argc, char* argv[]) {
    double *pMatrix; // Adjacency matrix
    TTreeNode** pMinSpanningTree; // Minimum spanning tree
    int Size; // Size of adjacency matrix

    // Process initialization
    ProcessInitialization(pMatrix, pMinSpanningTree, Size);

    // Serial Prim algorithm
    SerialPrim(pMatrix, pMinSpanningTree, Size);

    // Process termination
    ProcessTermination(pMatrix, pMinSpanningTree);
}
```

Поясним использование структур данных. Как уже отмечалось выше, для хранения минимального охватывающего дерева будет использоваться список примыканий, который представляет из себя массив, число элементов в котором совпадает с числом вершин в

графе; каждый  $i$ -ый элемент массива есть указатель на звено списка, описывающее вершину графа, соседнюю с вершиной  $i$ . Для описания вершины графа предназначена структура *TTreeNode*:

```
struct TTreeNode {
    int NodeNum;          // Vertex number
    double Distance;     // Weight of the edge
    TTreeNode* pNext;    // Link to the next node in list of adjacent vertices
};
```

**2. Функция ProcessInitialization.** Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества вершин в графе, выделения памяти для хранения матрицы смежности и для заполнения этой матрицы значениями. Начальные значения элементов матрицы смежности задаются в функции *RandomDataInitialization*.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double *&pMatrix, TTreeNode** &pMinSpanningTree,
                           int& Size) {
    do {
        printf("Enter the number of vertices: ");
        scanf("%d", &Size);
        if(Size <= 2)
            printf("The number of vertices should be greater than two\n");
    } while(Size <= 2);

    printf("Using graph with %d vertices\n", Size);

    // Allocate memory for the adjacency matrix
    pMatrix = new double[Size * Size];
    // Allocate memory for the spanning tree
    pMinSpanningTree = new TTreeNode* [Size];

    // Data initialization
    RandomDataInitialization(pMatrix, pMinSpanningTree, Size);
}
```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

**3. Функция SerialPrim.** Данная функция выполняет последовательный алгоритм Прима построения минимального охватывающего дерева.

```
// Function for serial Prim algorithm's execution
void SerialPrim(double *pMatrix, TTreeNode** pMinSpanningTree, int Size) {
    // Number of vertex that was added to the spanning tree on previous step
    int LastAdded;
    // Number of vertex which is nearest to the spanning tree
    TGraphNode NearestNode;
    // List of vertices, that haven't been added to minimum spanning tree
    TGraphNode **NotInMinSpanningTree = new TGraphNode* [Size-1];

    // Vertex with 0 number is the root of minimum spanning tree
    LastAdded = 0;
    // The other vertexes are not added to spanning tree
    for (int i=0; i<Size-1; i++) {
        NotInMinSpanningTree[i] = new TGraphNode;
        NotInMinSpanningTree[i]->NodeNum = i+1;
        NotInMinSpanningTree[i]->Distance = -1.0f;
        NotInMinSpanningTree[i]->ParentNodeNum = -1;
    }

    // Prim's algorithm iterations
```

```

for (int Iter=1; Iter<Size; Iter++) {
    // Recalculation of the distances
    for (int i=0; i<Size-1; i++) {
        if (NotInMinSpanningTree[i] != NULL) {
            double t1 = NotInMinSpanningTree[i]->Distance;
            double t2 =
                pMatrix[(NotInMinSpanningTree[i]->NodeNum)*Size+LastAdded];
            if (((t1<0) && (t2>0)) || ((t1>0) && (t2>0) && (t1>t2))) {
                NotInMinSpanningTree[i]->Distance = t2;
                NotInMinSpanningTree[i]->ParentNodeNum = LastAdded;
            }
        }
    }

    // Choose the nearest vertex
    NearestNode.NodeNum = -1;
    NearestNode.Distance = MAX_VALUE;
    for (int i=0; i<Size-1; i++) {
        if (NotInMinSpanningTree[i] != NULL) {
            double t1 = NotInMinSpanningTree[i]->Distance;
            double t2 = NearestNode.Distance;
            if ((t1>0) && (t1<t2)) {
                NearestNode.Distance = t1;
                NearestNode.NodeNum = NotInMinSpanningTree[i]->NodeNum;
            }
        }
    }

    // Add the nearest vertex to adjacency list,
    // which describes minimum spanning tree
    pMinSpanningTree[NearestNode.NodeNum] = new TTTreeNode;
    pMinSpanningTree[NearestNode.NodeNum]->NodeNum =
        NotInMinSpanningTree[NearestNode.NodeNum-1]->ParentNodeNum;
    pMinSpanningTree[NearestNode.NodeNum]->Distance = NearestNode.Distance;
    pMinSpanningTree[NearestNode.NodeNum]->pNext = NULL;

    int Parent = NotInMinSpanningTree[NearestNode.NodeNum-1]->ParentNodeNum;
    if (pMinSpanningTree[Parent] != NULL) {
        TTTreeNode *tmp = new TTTreeNode;
        tmp->pNext = pMinSpanningTree[Parent]->pNext;
        tmp->Distance = NearestNode.Distance;
        tmp->NodeNum = NearestNode.NodeNum;
        pMinSpanningTree[Parent]->pNext = tmp;
    }
    else {
        pMinSpanningTree[Parent] = new TTTreeNode;
        pMinSpanningTree[Parent]->pNext = NULL;
        pMinSpanningTree[Parent]->Distance = NearestNode.Distance;
        pMinSpanningTree[Parent]->NodeNum = NearestNode.NodeNum;
    }

    LastAdded = NearestNode.NodeNum;
    delete NotInMinSpanningTree[NearestNode.NodeNum-1];
    NotInMinSpanningTree[NearestNode.NodeNum-1] = NULL;
}

delete [] NotInMinSpanningTree;
}

```

Структура данных *TGraphNode* предназначена для описания вершины, не включенной в состав МОД. Каждый экземпляр этой структуры содержит номер вершины, текущее расстояние до МОД и номер вершины в МОД, смежной с ней:

```

struct TGraphNode {
    int NodeNum;           // Vertex number
    double Distance;       // Distance to the minimum spanning tree
    int ParentNodeNum;    // Number of vertex in minimum spanning tree,
                         // which is adjacent to the current vertex
};

```

### 10.2.3. Разделение вычислений на независимые части

Оценим возможности параллельного выполнения рассмотренного алгоритма нахождения минимального охватывающего дерева.

Итерации метода должны выполняться последовательно и, тем самым, не могут быть распараллелены. С другой стороны, выполняемые на каждой итерации алгоритма действия являются независимыми и могут реализовываться одновременно. Так, например, определение величин  $d_i$  может осуществляться для каждой вершины графа в отдельности, нахождение дуги минимального веса может быть реализовано по каскадной схеме и т.д.

Распределение данных между вычислительными элементами должно обеспечивать независимость перечисленных операций алгоритма Прима.

Общая схема параллельного выполнения алгоритма Прима будет состоять в следующем:

- определяется вершина  $t$  графа  $G$ , имеющая дугу минимального веса до множества  $V_T$ ; для выбора такой вершины необходимо осуществить поиск минимума в наборах величин  $d_i$ , имеющихся на каждом из вычислительных элементов, и выполнить редукцию полученных значений;
- номер выбранной вершины для включения в охватывающее дерево передается всем вычислительным элементам;
- обновляются наборы величин  $d_i$  с учетом добавления новой вершины.

### 10.2.4. Анализ эффективности параллельных вычислений

Как и ранее, при анализе эффективности параллельного алгоритма Прима будем предполагать, что время выполнения складывается из времени вычислений (которые могут быть выполнены вычислительными элементами параллельно) и времени, необходимого на загрузку необходимых данных из оперативной памяти в кэш. Доступ к памяти осуществляется строго последовательно.

Для выполнения алгоритма Прима над графом, содержащим  $n$  вершин, требуется выполнение  $n$  итераций алгоритма, на каждой из которых параллельно выполняется пересчет расстояний от вершин, не входящих в состав МОД, до МОД и выбор вершины, расстояние от которой минимально. Следовательно, на каждой  $i$ -ой итерации алгоритма выполняется  $(n-i)$  операций выбора минимума для пересчета расстояний и  $(n-i)$  операций сравнения для выбора ближайшей вершины (кроме того, после выбора потоками «локальных» ближайших вершин необходимо выполнить редукцию полученных значений для получения «глобальной» ближайшей вершины). Таким образом, для оценки времени выполнения вычислений может быть использовано соотношение:

$$T_p(\text{calc}) = \left( 2 \cdot \frac{\sum_{i=0}^{n-1} (n-i)}{p} + \log_2 p \right) \cdot t = \left( 2 \cdot \frac{n^2/2}{p} + \log_2 p \right) \cdot t = \left( \frac{n^2}{p} + \log_2 p \right) \cdot t, \quad (10.6)$$

где  $t$  есть время выполнения операции выбора минимального значения.

Если количество вершин в исходном графе настолько велико, что описывающая граф матрица смежности не может быть полностью помещена в кэш, то на каждой итерации внешнего цикла происходит считывание из оперативной памяти в кэш вычислительных элементов необходимых значений. Так, для пересчета расстояний от вершин, не включенных в состав МОД, до МОД, необходимо знать расстояние от этих вершин до вершины, включенной в состав МОД на последнем шаге. Следовательно, на  $i$ -ой итерации алгоритма Прима необходимо загрузить из оперативной памяти  $(n-i)$  значений. Каждое значение – это число с расширенной точностью типа `double`, то есть занимает 8 байт. Следует напомнить, что данные из оперативной памятичитываются строками по 64 байта. Таким образом, затраты на доступ к памяти составляют:

$$T_p(\text{mem}) = \sum_{i=0}^{n-1} \frac{64 \cdot (n-i)}{b} = 64 \cdot \frac{n^2/2}{b} \quad (10.7)$$

Также следует учесть латентность оперативной памяти  $\alpha$ :

$$T_p(\text{mem}) = (n^2/2) \cdot \left( \alpha + \frac{64}{b} \right) \quad (10.8)$$

Необходимо отметить, что при выборе вершины, находящейся на наименьшем расстоянии от минимального охватывающего дерева, и при добавлении новой вершины в состав МОД, используются структуры данных сравнительно небольшого размера, которые сохраняются в кэш при переходе от одной итерации алгоритма Прима к другой.

При получении итоговой оценки времени выполнения параллельного алгоритма Прима необходимо также учитывать затраты на организацию и закрытие параллельных секций. На каждой итерации алгоритма создается две параллельные секции: первая отвечает за параллельное выполнение пересчета расстояний, а вторая – за выбор ближайшей вершины:

$$T_p = \left( \frac{n^2}{p} + \log_2 p \right) t + (n^2/2) \cdot \left( \alpha + \frac{64}{b} \right) + 2nd, \quad (10.9)$$

где  $\delta$  есть накладные расходы на организацию параллельности на каждой итерации алгоритма.

Для улучшения точности модели необходимо учесть частоту кэш промахов  $\gamma$  (см. п. 6.5.4):

$$T_p = \left( \frac{n^2}{p} + \log_2 p \right) t + g(n^2/2) \cdot \left( \alpha + \frac{64}{b} \right) + 2nd, \quad (10.10)$$

### 10.2.5. Программная реализация параллельного алгоритма Прима

Представим возможный вариант программы, выполняющей параллельный алгоритм Прима построения минимального охватывающего дерева.

Для распараллеливания этапа пересчета расстояний от вершин, не включенных в состав минимального охватывающего дерева, до МОД, достаточно распределить итерации цикла, просматривающего все элементы массива `NotInSpanningTree`, между потоками параллельной программы при помощи директивы `parallel for`. Однако, поскольку некоторые элементы этого массива могут быть пустыми и, следовательно, не подлежат обработке, то для лучшей балансировки вычислительной нагрузки вычислительных элементов следует использовать динамическое планирование.

Для выбора вершины, ближайшей к уже построенной части МОД, применим следующий подход. В каждом потоке заведем локальную переменную `ThreadNearestNode` для хранения «локальной» ближайшей вершины. Выбор «локальных» ближайших вершин выполняется потоками параллельно. Далее выполняется редукция полученных значений при помощи механизма критических секций.

```

// Function for parallel Prim algorithm's execution
void ParallelPrim(double *pMatrix, TTreeNode** pMinSpanningTree, int Size) {
    // Number of vertex that was added to the spanning tree on previous step
    int LastAdded;
    // Number of vertex which is nearest to the spanning tree
    TGraphNode NearestNode;
    // List of vertices, that haven't been added to minimum spanning tree
    TGraphNode **NotInMinSpanningTree = new TGraphNode* [Size-1];

    // Vertex with 0 number is the root of minimum spanning tree
    LastAdded = 0;
    // The other vertexes are not added to spanning tree
    for (int i=0; i<Size-1; i++)
    {
        NotInMinSpanningTree[i] = new TGraphNode;
        NotInMinSpanningTree[i]->NodeNum = i+1;
        NotInMinSpanningTree[i]->Distance = -1.0f;
        NotInMinSpanningTree[i]->ParentNodeNum = -1;
    }

    // Prim's algorithm iterations
    for (int Iter=1; Iter<Size; Iter++)
    {
        // Recalculation of the distances
#pragma omp parallel for schedule (dynamic,1)
        for (int i=0; i<Size-1; i++) {
            if (NotInMinSpanningTree[i] != NULL) {
                double t1 = NotInMinSpanningTree[i]->Distance;
                double t2 =
                    pMatrix[(NotInMinSpanningTree[i]->NodeNum)*Size+LastAdded];
                if (((t1<0) && (t2>0)) || ((t1>0) && (t2>0) && (t1>t2))) {
                    NotInMinSpanningTree[i]->Distance = t2;
                    NotInMinSpanningTree[i]->ParentNodeNum = LastAdded;
                }
            }
        }

        // Choose the nearest vertex
        NearestNode.NodeNum = -1;
        NearestNode.Distance = MAX_VALUE;
#pragma omp parallel
        {
            TGraphNode ThreadNearestNode;
            ThreadNearestNode.NodeNum = -1;
            ThreadNearestNode.Distance = MAX_VALUE;
#pragma omp for schedule (dynamic,1)
            for (int i=0; i<Size-1; i++) {
                if (NotInMinSpanningTree[i] != NULL) {
                    double t1 = NotInMinSpanningTree[i]->Distance;
                    double t2 = ThreadNearestNode.Distance;
                    if ((t1>0) && (t1<t2)) {
                        ThreadNearestNode.Distance = t1;
                        ThreadNearestNode.NodeNum = NotInMinSpanningTree[i]->NodeNum;
                    }
                }
            } // for
#pragma omp critical
            {
                if (ThreadNearestNode.Distance < NearestNode.Distance) {
                    NearestNode.Distance = ThreadNearestNode.Distance;
                    NearestNode.NodeNum = ThreadNearestNode.NodeNum;
                }
            } // pragma omp critical
        }
    }
}

```

```

} // pragma omp parallel

// Add the nearest vertex to adjacency list,
// which describes minimum spanning tree
pMinSpanningTree[NearestNode.NodeNum] = new TTTreeNode;
pMinSpanningTree[NearestNode.NodeNum]->NodeNum =
    NotInMinSpanningTree[NearestNode.NodeNum-1]->ParentNodeNum;
pMinSpanningTree[NearestNode.NodeNum]->Distance = NearestNode.Distance;
pMinSpanningTree[NearestNode.NodeNum]->pNext = NULL;

int Parent = NotInMinSpanningTree[NearestNode.NodeNum-1]->ParentNodeNum;
if (pMinSpanningTree[Parent] != NULL) {
    TTTreeNode *tmp = new TTTreeNode;
    tmp->pNext = pMinSpanningTree[Parent]->pNext;
    tmp->Distance = NearestNode.Distance;
    tmp->NodeNum = NearestNode.NodeNum;
    pMinSpanningTree[Parent]->pNext = tmp;
}
else {
    pMinSpanningTree[Parent] = new TTTreeNode;
    pMinSpanningTree[Parent]->pNext = NULL;
    pMinSpanningTree[Parent]->Distance = NearestNode.Distance;
    pMinSpanningTree[Parent]->NodeNum = NearestNode.NodeNum;
}

LastAdded = NearestNode.NodeNum;
delete NotInMinSpanningTree[NearestNode.NodeNum-1];
NotInMinSpanningTree[NearestNode.NodeNum-1] = NULL;
}
delete [] NotInMinSpanningTree;
}

```

#### 10.2.6. Результаты вычислительных экспериментов

Эксперименты проводились на двухпроцессорном вычислительном узле на базе четырехядерных процессоров Intel Xeon E5320, 1,86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

Результаты вычислительных экспериментов приведены в таблице 10.4. Времена выполнения алгоритмов указаны в секундах.

**Таблица 10.4.** Результаты вычислительных экспериментов для параллельного алгоритма Прима  
(при использовании двух и четырех вычислительных ядер)

количество вершин в графе	Последовательный алгоритм	Параллельный алгоритм			
		2 потока		4 потока	
		время	ускорение	время	ускорение
100	0,0005	0,0009	0,5351	0,0009	0,5333
200	0,0017	0,0024	0,7209	0,0020	0,8487
300	0,0038	0,0038	0,9942	0,0037	1,0148
400	0,0068	0,0059	1,1392	0,0048	1,3983
500	0,0109	0,0088	1,2367	0,0070	1,5501
600	0,0167	0,0120	1,3934	0,0089	1,8820
700	0,0236	0,0160	1,4696	0,0114	2,0725
800	0,0332	0,0208	1,5956	0,0144	2,3017
900	0,0467	0,0256	1,8257	0,0177	2,6315

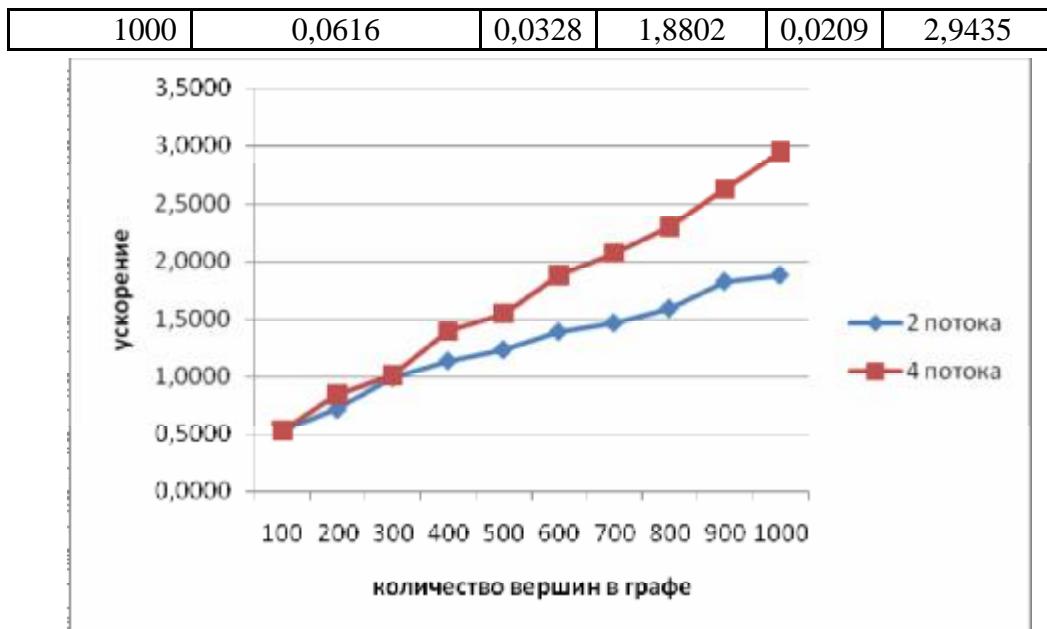


Рис. 10.8. Зависимость ускорения от количества исходных данных при выполнении параллельного метода Прима

В таблице 10.5 и 10.6 и на рис. 10.9 и 10.9 представлены результаты сравнения времени выполнения параллельного метода Прима с использованием двух и четырех потоков со временем, полученным при помощи моделей (10.9) и (10.10).

Для того, чтобы оценить время одной операции выбора минимального значения  $\tau$ , воспользуемся уже известным подходом. Измерим время выполнения последовательного алгоритма Прима при малых объемах данных, таких, чтобы матрица смежности, описывающая граф, могла быть полностью помещена в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним матрицу случайными значениями. Выполнение этого действия гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение  $\tau$ , равное 47,5 нс. Как и ранее, латентность  $\alpha$  и пропускная способность канала доступа к оперативной памяти  $\beta$  являются равными  $\alpha = 8,31$  нс и  $\beta = 12,44$  Гб/с.

Частота кэш промахов, измеренная с помощью системы VPS, для двух потоков оказалась равной 0,5656, а для четырех потоков значение этой величины была оценена как 0,6614. Как отмечалось в главе 6, время  $\delta$ , необходимое на организацию и закрытие параллельной секции, составляет 0,25 мкс.

**Таблица 10.5.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Прима с использованием двух потоков

Размер массива	$T_p$	$T_p^* \text{ (calc)}$ (модель)	Модель 10.9 – оценка сверху		Модель 10.10 – уточненная оценка	
			$T_p^* \text{ (mem)}$	$T_p^*$	$T_p^* \text{ (mem)}$	$T_p^*$
100	0,0009	0,0003	0,0001	0,0004	0,0000	0,0003
200	0,0024	0,0011	0,0003	0,0013	0,0001	0,0012

300	0,0038	0,0023	0,0006	0,0029	0,0003	0,0026
400	0,0059	0,0040	0,0010	0,0050	0,0006	0,0046
500	0,0088	0,0062	0,0016	0,0078	0,0009	0,0071
600	0,0120	0,0089	0,0024	0,0112	0,0013	0,0102
700	0,0160	0,0120	0,0032	0,0152	0,0018	0,0138
800	0,0208	0,0156	0,0042	0,0198	0,0024	0,0180
900	0,0256	0,0197	0,0053	0,0250	0,0030	0,0227
1000	0,0328	0,0243	0,0066	0,0308	0,0037	0,0280

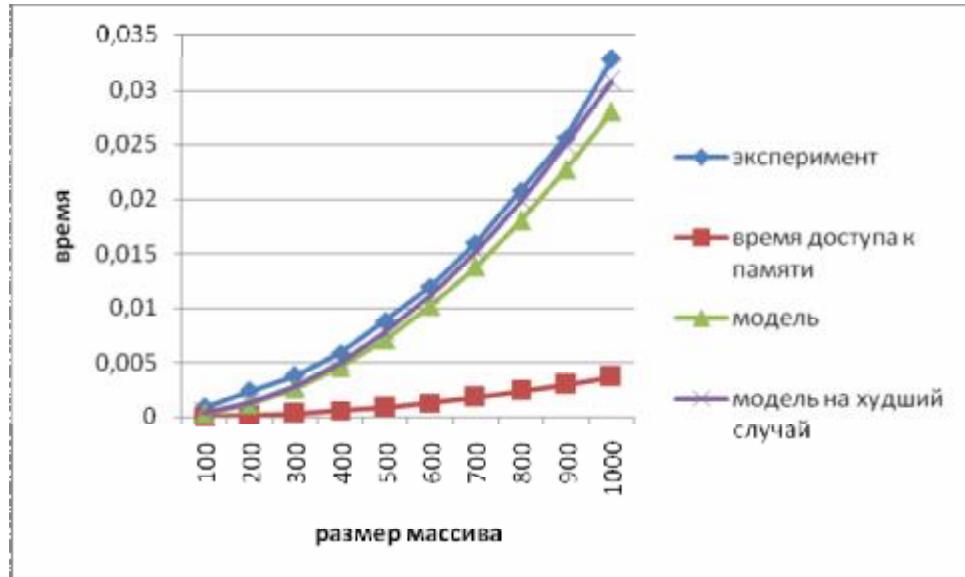


Рис. 10.9. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Прима от объема исходных данных при использовании двух потоков

**Таблица 10.6.** Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Прима с использованием четырех потоков

Размер массива	$T_p$	$T_p^* (calc)$ (модель)	Модель 10.9 – оценка сверху		Модель 10.10 – уточненная оценка	
			$T_p^* (tem)$	$T_p^*$	$T_p^* (tem)$	$T_p^*$
100	0,0009	0,0002	0,0001	0,0002	0,0000	0,0002
200	0,0020	0,0006	0,0003	0,0008	0,0002	0,0007
300	0,0037	0,0012	0,0006	0,0018	0,0004	0,0016
400	0,0048	0,0021	0,0010	0,0031	0,0007	0,0028
500	0,0070	0,0032	0,0016	0,0049	0,0011	0,0043
600	0,0089	0,0046	0,0024	0,0069	0,0016	0,0061
700	0,0114	0,0062	0,0032	0,0094	0,0021	0,0083
800	0,0144	0,0080	0,0042	0,0122	0,0028	0,0108
900	0,0177	0,0101	0,0053	0,0154	0,0035	0,0136
1000	0,0209	0,0124	0,0066	0,0189	0,0043	0,0167

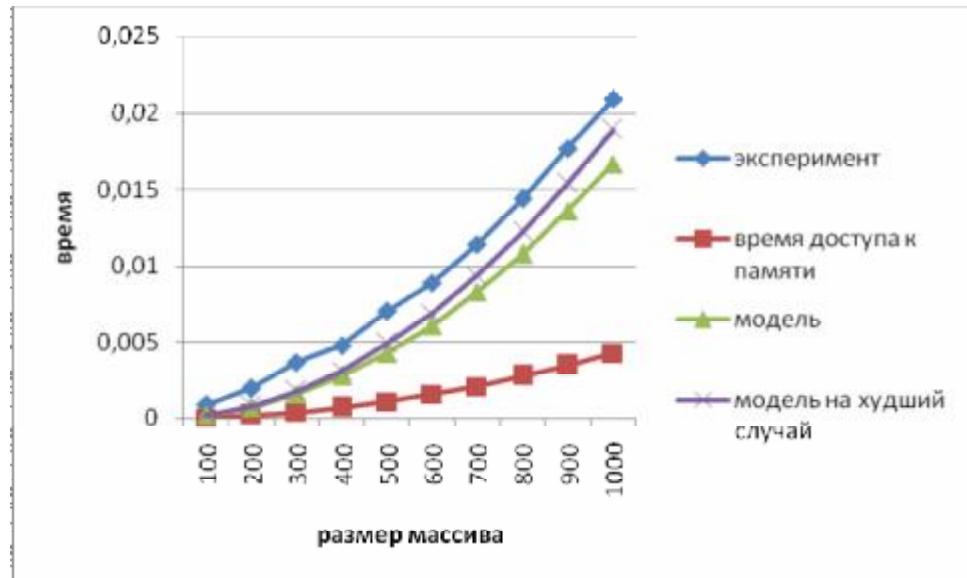


Рис. 10.10. График зависимости экспериментального и теоретического времени выполнения параллельного алгоритма Прима от объема исходных данных при использовании четырех потоков

### 10.3. Задача оптимального разделения графов

Проблема оптимального разделения графов относится к числу часто возникающих задач при проведении различных научных исследований, использующих параллельные вычисления. В качестве примера можно привести задачи обработки данных, в которых области расчетов представляются в виде двухмерной или трехмерной сети. Вычисления в таких задачах сводятся, как правило, к выполнению тех или иных процедур обработки для каждого элемента (узла) сети. При этом в ходе вычислений между соседними элементами сети может происходить передача результатов обработки и т.п. Эффективное решение таких задач на многопроцессорных системах с распределенной памятью или вычислительных системах с общей памятью, в которых вычислительные элементы имеют раздельный кэш, предполагает разделение сети между вычислительными элементами таким образом, чтобы каждому из вычислительных элементов выделялось примерно равное число элементов сети, а межпроцессорные коммуникации, необходимые для выполнения информационного обмена между соседними элементами, были минимальными. На рис. 10.11. показан пример нерегулярной сети, разделенной на 4 части (различные части разбиения сети выделены темным цветом различной интенсивности).

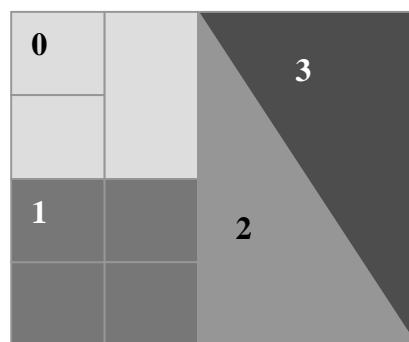


Рис. 10.11. Пример разделения нерегулярной сети

Очевидно, что такие задачи разделения сети между процессорами могут быть сведены к проблеме оптимального разделения графа. Данный подход целесообразен еще и потому, что представление модели вычислений в виде графа позволяет легче решить вопросы хранения обрабатываемых данных и предоставляет возможность применения типовых алгоритмов обработки графов.

Для представления сети в виде графа каждому элементу сети можно поставить в соответствие вершину графа, а дуги графа использовать для отражения свойства близости элементов сети (например, определять дуги между вершинами графа тогда и только тогда, когда соответствующие элементы исходной сети являются соседними). При таком подходе, например, для сети на рис. 10.11, будет сформирован график, приведенный на рис. 10.12.

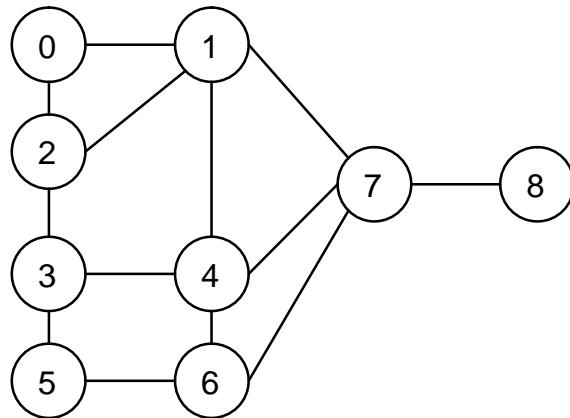


Рис. 10.12. Пример графа, моделирующего структуру сети на рис. 10.11

Дополнительная информация по проблеме разделения графов может быть получена, например, в [90].

Задача оптимального разделения графов сама может являться предметом распараллеливания. Это бывает необходимо в тех случаях, когда вычислительной мощности и объема оперативной памяти обычных компьютеров недостаточно для эффективного решения задачи. Параллельные алгоритмы разделения графов рассматриваются во многих научных работах – см., например, [41-42, 59, 69-70, 87, 98].

### 10.3.1. Постановка задачи оптимального разделения графов

Пусть дан взвешенный неориентированный граф  $G = (V, E)$ , каждой вершине  $v \in V$  и каждому ребру  $e \in E$  которого приписан вес. Задача оптимального разделения графа состоит в разбиении его вершин на непересекающиеся подмножества с максимально близкими суммарными весами вершин и минимальным суммарным весом ребер, проходящих между полученными подмножествами вершин.

Следует отметить возможную противоречивость указанных критериев разбиения графа – равновесность подмножеств вершин может не соответствовать минимальности весов граничных ребер и наоборот. В большинстве случаев необходимым является выбор того или иного компромиссного решения. Так, в случае невысокой доли коммуникаций может оказаться эффективным оптимизировать вес ребер только среди решений, обеспечивающих оптимальное разбиение множества вершин по весу.

Далее для простоты изложения учебного материала будем полагать веса вершин и ребер графа равными единице.

### 10.3.2. Метод рекурсивного деления пополам

Для решения задачи разбиения графа можно рекурсивно применить *метод бинарного деления*, при котором на первой итерации граф разделяется на две равные части, далее на втором шаге каждая из полученных частей также разбивается на две части и т.д. В данном подходе для разбиения графа на  $k$  частей необходимо  $\log_2 k$  уровней рекурсии и выполнение  $k-1$  деления пополам. В случае, когда требуемое количество разбиений  $k$  не является степенью двойки, каждое деление пополам необходимо осуществлять в соответствующем соотношении.

Поясним схему работы метода деления пополам на примере разделения графа на рис. 10.12 на 5 частей. Сначала граф следует разделить на 2 части в соотношении 2:3 (непрерывная линия), затем правую часть разбиения в отношении 1:3 (пунктирная линия), после этого осталось разделить 2 крайние подобласти слева и справа в отношении 1:1 (пунктир с точкой).

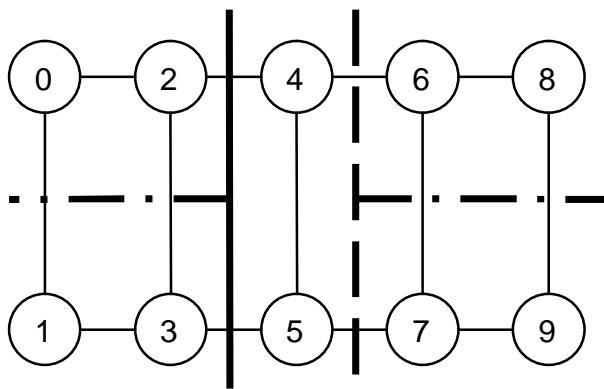


Рис. 10.13. Пример разбиения графа на 5 частей методом рекурсивного деления пополам

### 10.3.3. Геометрические методы

Геометрические методы (см., например, [43, 58, 65, 73, 75, 77, 81, 86]) выполняют разбиение сетей, основываясь исключительно на координатной информации об узлах сети. Так как эти методы не принимают во внимание информацию о связности элементов сети, то они не могут явно привести к минимизации суммарного веса граничных ребер (в терминах графа, соответствующего сети). Для минимизации межпроцессорных коммуникаций геометрические методы оптимизируют некоторые вспомогательные показатели (например, длину границы между разделенными участками сети).

Обычно геометрические методы не требуют большого объема вычислений, однако качество их разбиения обычно уступает методам, принимающим во внимание связность элементов сети.

**1. Покоординатное разбиение.** *Покоординатное разбиение* (*coordinate nested dissection*) – это метод, основанный на рекурсивном делении пополам сети по наиболее длинной стороне. В качестве иллюстрации рис. 10.13 показан пример сети, при разделении которой именно такой способ разбиения дает существенно меньшее количество информационных связей между разделенными частями, по сравнению со случаем, когда сеть делится по меньшей (вертикальной) стороне.

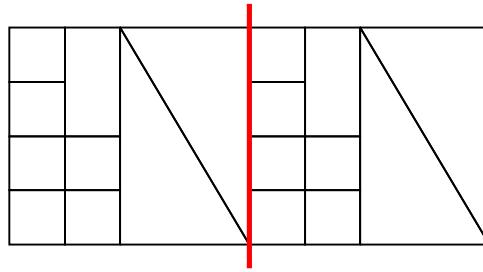


Рис. 10.14. Пример разделения сети графическим методом по наибольшей размерности (граница раздела показана жирной линией)

Общая схема выполнения метода состоит в следующем. Сначала вычисляются центры масс элементов сети. Полученные точки проектируются на ось, соответствующую наибольшей стороне разделяемой сети. Таким образом, мы получаем упорядоченный список всех элементов сети. Делением списка пополам (возможно, в нужной пропорции) мы получаем требуемую бисекцию. Аналогичным способом полученные фрагменты разбиения рекурсивно делятся на нужное число частей.

Метод координатного вложенного разбиения работает очень быстро и требует небольшого количества оперативной памяти. Однако, получаемое разбиение уступает по качеству более сложным и вычислительно-трудоемким методам. Кроме того, в случае сложной структуры сети алгоритм может получать разбиение с несвязанными подсетями.

**2. Рекурсивный инерционный метод деления пополам.** Предыдущая схема могла производить разбиение сети только по линии, перпендикулярной одной из координатных осей. Во многих случаях такое ограничение оказывается критичным для построения качественного разбиения. Достаточно повернуть сеть на рис. 10.14 под острым углом к координатным осям (см. рис. 10.15), чтобы убедиться в этом. Для минимизации границы между подсетями желательна возможность проведения линии разделения с любым требуемым углом поворота. Возможный способ определения угла поворота, используемый в *рекурсивном инерционном методе деления пополам (recursive inertial bisection)*, состоит в использовании главной инерционной оси (см., например, [84]), считая элементы сети точечными массами. Линия бисекции, ортогональная полученной оси, как правило, дает границу наименьшей длины.

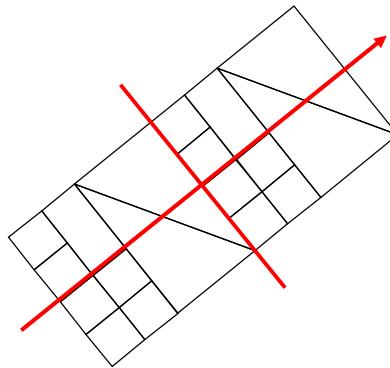


Рис. 10.15. Пример разделения сети методом рекурсивной инерционной бисекции (стрелкой показана главная инерционная ось)

**3. Деление сети с использованием кривых Пеано.** Одним из недостатков предыдущих графических методов является то, что при каждой бисекции эти методы учитывают только одну размерность. Таким образом, схемы, учитывающие больше размерностей, могут обеспечить лучшее разбиение.

Один из таких методов упорядочивает элементы в соответствии с позициями центров их масс вдоль кривых Пеано. Кривые Пеано – это кривые, полностью заполняющие

фигуры больших размерностей (например, квадрат или куб). Применение таких кривых обеспечивает близость точек фигуры, соответствующих точкам, близким на кривой. После получения списка элементов сети, упорядоченного в соответствии с расположением на кривой, достаточно разделить список на необходимое число частей в соответствии с установленным порядком. Получаемый в результате такого подхода метод носит в литературе наименование *алгоритма деления сети с использованием кривых Пеано* (*space-filling curve technique*). Подробнее о методе можно прочитать в работах [76-77,81].

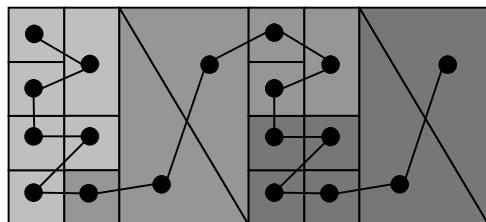


Рис. 10.16. Пример разделения сети на 3 части с использованием кривых Пеано

#### 10.3.4. Комбинаторные методы

В отличие от геометрических методов, комбинаторные алгоритмы (см., например, [57,90]) обычно оперируют не с сетью, а с графом, построенным для этой сети. Соответственно, в отличие от геометрических схем комбинаторные методы не принимают во внимание информацию о близости расположения элементов сети друг относительно друга, руководствуясь только смежностью вершин графа. Комбинаторные методы обычно обеспечивают более сбалансированное разбиение и меньшее информационное взаимодействие полученных подсетей. Однако комбинаторные методы имеют тенденцию работать существенно дольше, чем их геометрические аналоги.

**1. Деление с учетом связности.** С самых общих позиций понятно, что при разделении графа информационная зависимость между разделенными подграфами будет меньше, если соседние вершины (вершины, между которыми имеются дуги) будут находиться в одном подграфе. Алгоритм деления графов с учетом связности (*levelized nested dissection*) пытается достичь этого, последовательно добавляя к формируемому подграфу соседей. На каждой итерации алгоритма происходит разделение графа на 2 части. Таким образом, разделение графа на требуемое число частей достигается путем рекурсивного применения алгоритма.

Общая схема алгоритма может быть описана при помощи следующего набора правил.

1. `Iteration = 0`
2. Присвоение номера `Iteration` произвольной вершине графа
3. Присвоение ненумерованным соседям вершин с номером `Iteration` номера `Iteration + 1`
4. `Iteration = Iteration + 1`
5. Если еще есть неперенумерованные соседи, то переход на шаг 3
6. Разделение графа на 2 части в порядке нумерации

#### Алгоритм 10.2.      Общая схема выполнения алгоритма деления графов с учетом связности

Для минимизации информационных зависимостей имеет смысл в качестве начальной выбирать граничную вершину. Поиск такой вершины можно осуществить методом, близким к рассмотренной схеме. Так, перенумеровав вершины графа в соответствии с

алгоритмом 10.2 (начиная нумерацию из произвольной вершины), мы можем взять любую вершину с максимальным номером. Как нетрудно убедиться, она будет граничной.

Пример работы алгоритма приведен на рис. 10.17. Цифрами показаны номера, которые получили вершины в процессе разделения. Сплошной линией показана граница, разделяющая 2 подграфа. Также на рисунке показано лучшее решение (пунктирная линия). Очевидно, что полученное алгоритмом разбиение далеко от оптимального, так как в приведенном примере есть решение только с 3 пересеченными ребрами вместо 5.

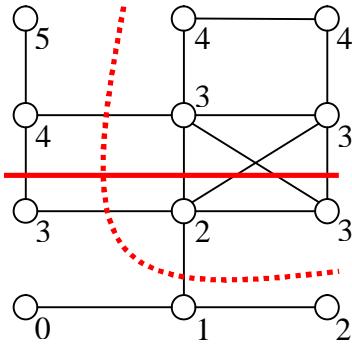


Рис. 10.17. Пример работы алгоритма деления графов с учетом связности

**2. Алгоритм Кернигана-Лина.** В алгоритме Кернигана-Лина (*Kernighan-Lin algorithm*) используется несколько иной подход для решения проблемы оптимального разбиения графа – при начале работы метода предполагается, что некоторое начальное разбиение графа уже существует, затем же имеющееся приближение улучшается в течение некоторого количества итераций. Используемый способ улучшения в алгоритме Кернигана-Лина состоит в обмене вершинами между подмножествами имеющегося разбиения графа (см. рис. 10.18). Для формирования требуемого количества частей графа может быть использована, как и ранее, рекурсивная процедура деления пополам.

Общая схема одной итерации алгоритма Кернигана-Лина может быть представлена следующим образом.

#### 1. Формирование множества пар вершин для перестановки

Из вершин, которые еще не были переставлены на данной итерации, формируются все возможные пары (в парах должны присутствовать по одной вершине из каждой части имеющегося разбиения графа).

#### 2. Построение новых вариантов разбиения графа

Каждая пара, подготовленная на шаге 1, поочередно используется для обмена вершин между частями имеющегося разбиения графа для получения множества новых вариантов деления.

#### 3. Выбор лучшего варианта разбиения графа

Для сформированного на шаге 2 множества новых делений графа выбирается лучший вариант. Данный способ фиксируется как новое текущее разбиение графа, а соответствующая выбранному варианту пары вершин отмечается как использованная на текущей итерации алгоритма.

#### 4. Проверка использования всех вершин

При наличии в графе вершин, еще неиспользованных при перестановках, выполнение итерации алгоритма снова продолжается с шага 1. Если же перебор вершин графа завершен, далее следует шаг 5.

#### 5. Выбор наилучшего варианта разбиения графа

Среди всех разбиений графа, полученных на шаге 3 проведенных итераций, выбирается (и фиксируется) наилучший вариант разбиения графа.

Поясним дополнительно, что на шаге 2 итерации алгоритма перестановка вершин каждой очередной пары осуществляется для одного и того разбиения графа, выбранного до начала выполнения итерации или определенного на шаге 3. Общее количество выполняемых итераций, как правило, фиксируется заранее и является параметром алгоритма (за исключением случая остановки при отсутствии улучшения разбиения на очередной итерации).

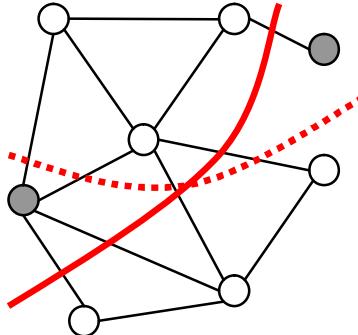


Рис. 10.18. Пример перестановки двух вершин (выделены серым) в методе Кернигана-Лина

### 10.3.5. Сравнение алгоритмов разбиения графов

Рассмотренные алгоритмы разбиения графов различаются точностью получаемых решений, временем выполнения и возможностями для распараллеливания (под точностью понимается величина близости получаемых при помощи алгоритмов решений к оптимальным вариантам разбиения графов). Выбор наиболее подходящего алгоритма в каждом конкретном случае является достаточно сложной и неочевидной задачей. Проведению такого выбора может содействовать сведенная воедино в табл. 10.7 (см. [90]) общая характеристика ряда алгоритмов разделения графов, рассмотренных в данном разделе. Дополнительная информация по проблеме оптимального разбиения графов может быть получена, например, в [90].

**Таблица 10.7.** Сравнительная таблица некоторых алгоритмов разделения графов

	Необходимость координатной информации	Точность	Время выполнения	Возможности для распараллеливания
Покоординатное разбиение	да	•	•	•••
Рекурсивный инерционный метод деления пополам	да	••	•	•••
Деление с учетом связности	нет	••	••	••
Алгоритм Кернигана-Лина	1 итерация	нет	••	•
	10 итераций	нет	••••	••
	50 итераций	нет	•••••	••

Столбец "Необходимость координатной информации" отмечает использование алгоритмом координатной информации об элементах сети или вершинах графа.

Столбец "Точность" дает качественную характеристику величины приближения получаемых алгоритмом решений к оптимальным вариантам разбиения графов. Каждый

дополнительный закрашенный кружок определяет примерно 10%-процентное улучшение точности приближения (соответственно, заштрихованный наполовину кружок означает 5%-процентное улучшение получаемого решения).

Столбец "Время выполнения" показывает относительное время, затрачиваемое различными алгоритмами разбиения. Каждый дополнительный закрашенный кружок соответствует увеличению времени разбиения примерно в 10 раз (заштрихованный наполовину кружок отвечает за увеличение времени разбиения примерно в 5 раз).

Столбец "Возможности для распараллеливания" характеризует свойства алгоритмов для параллельного выполнения. Алгоритм Кернигана-Лина при выполнении только одной итерации почти не поддается распараллеливанию. Этот же алгоритм при большем количестве итераций, а также метод деления с учетом связности, могут быть распараллелены со средней эффективностью. Алгоритм покоординатного разбиения и рекурсивный инерционный метод деления пополам обладают высокими показателями для распараллеливания.

## 10.4. Краткий обзор главы

В главе рассмотрены ряд алгоритмов для решения типовых задач обработки графов. Кроме того, в разделе приведен обзор методов разделения графа.

В 10.1 представлен *алгоритм Флойда (Floyd)* для решения задачи поиска путей минимальной длины между каждой парой вершин графа. Для алгоритма дается общая вычислительная схема последовательного варианта метода, обсуждаются способы его распараллеливания, проводится анализ эффективности получаемых параллельных вычислений, рассматривается программная реализация метода и приводятся результаты вычислительных экспериментов.

В 10.2 рассматривается *алгоритм Прима (Prim)* для решения задачи поиска минимального охватывающего дерева (остова) неориентированного взвешенного графа. Остовом графа называют связный подграф без циклов (дерево), содержащий все вершины исходного графа и ребра, имеющие минимальный суммарный вес. Для алгоритма дается общее описание его исходного последовательного варианта, определяются возможные способы его параллельного выполнения, определяются теоретические оценки параллельных вычислений, рассматриваются результаты проведенных вычислительных экспериментов.

Рассматриваемая в 10.3 задача оптимального разделения графов является важной для многих научных исследований, использующих параллельные вычисления. Для примера в подразделе приведен общий способ перехода от двухмерной или трехмерной сети, моделирующей процесс вычислений, к соответствующему ей графу. Для решения задачи разбиения графов были рассмотрены *геометрические методы*, использующие при разделении сетей только координатную информацию об узлах сети, и *комбинаторные алгоритмы*, руководствуясь смежностью вершин графа. К числу рассмотренных геометрических методов относятся *покоординатное разбиение (coordinate nested dissection)*, *рекурсивный инерционный метод деления пополам (recursive inertial bisection)*, *деление сети с использованием кривых Пеано (space-filling curve techniques)*. К числу рассмотренных комбинаторных алгоритмов относятся *деление с учетом связности (levelized nested dissection)* и *алгоритм Кернигана-Лина (Kernighan-Lin algorithm)*. Для сопоставления рассмотренных подходов приводится общая сравнительная характеристика алгоритмов по времени выполнения, точности получаемого решения, возможностей для распараллеливания и т.п.

## **10.5. Обзор литературы**

Дополнительная информация по алгоритмам Флойда и Прима может быть получена, например, в [17].

Подробное рассмотрение вопросов, связанных с проблемой разделения графов, содержится в работах [43,57-58,65,73,75,77,81,86,90].

Параллельные алгоритмы разделения графов рассматриваются в [41,58,65,69-70,87,98].

## **10.6. Контрольные вопросы**

1. Приведите определение графа. Какие основные способы используются для задания графов?
2. В чем состоит задача поиска всех кратчайших путей?
3. Приведите общую схему алгоритма Флойда. Какова трудоемкость алгоритма?
4. В чем состоит способ распараллеливания алгоритма Флойда?
5. В чем заключается задача нахождения минимального охватывающего дерева? Приведите пример использования задачи на практике.
6. Приведите общую схему алгоритма Прима. Какова трудоемкость алгоритма?
7. В чем состоит способ распараллеливания алгоритма Прима?
8. В чем отличие геометрических и комбинаторных методов разделения графа? Какие методы являются более предпочтительными? Почему?
9. Приведите описание метода поординатного разбиения и алгоритма разделения с учетом связности. Какой из этих методов является более простым для реализации?

## **10.7. Задачи и упражнения**

1. Используя приведенный программный код, выполните реализацию параллельного алгоритма Флойда. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом параметров используемой вычислительной системы. Сравните полученные оценки с экспериментальными данными.
2. Выполните реализацию параллельного алгоритма Прима. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом параметров используемой вычислительной системы. Сравните полученные оценки с экспериментальными данными.
3. Разработайте программную реализацию алгоритма Кернигана – Лина. Дайте оценку возможности распараллеливания этого алгоритма.

Глава 11. Решение дифференциальных уравнений в частных производных .....	1
11.1. Последовательные методы решения задачи Дирихле.....	2
11.2. Организация параллельных вычислений для систем с общей памятью .....	4
11.2.1. Использование OpenMP для организации параллелизма .....	4
11.2.2. Проблема синхронизации параллельных вычислений.....	5
11.2.3. Возможность неоднозначности вычислений в параллельных программах .....	8
11.2.4. Проблема взаимоблокировки .....	9
11.2.5. Исключение неоднозначности вычислений.....	10
11.2.6. Волновые схемы параллельных вычислений.....	12
11.2.7. Балансировка вычислительной нагрузки процессоров .....	17
11.3. Организация параллельных вычислений для систем с распределенной памятью .....	18
11.3.1. Разделение данных .....	18
11.3.2. Обмен информацией между процессорами .....	19
11.3.3. Коллективные операции обмена информацией.....	22
11.3.4. Организация волны вычислений .....	23
11.3.5. Блочная схема разделения данных.....	24
11.3.6. Оценка трудоемкости операций передачи данных .....	27
11.4. Краткий обзор главы .....	28
11.5. Обзор литературы .....	29
11.6. Контрольные вопросы .....	29
11.7. Задачи и упражнения.....	29

## Глава 11.

### Решение дифференциальных уравнений в частных производных

Дифференциальные уравнения в частных производных представляют собой широко применяемый математический аппарат при разработке моделей в самых разных областях науки и техники. К сожалению, явное решение этих уравнений в аналитическом виде оказывается возможным только в частных простых случаях, и, как результат, возможность анализа математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения. Объем выполняемых при этом вычислений обычно является значительным и использование высокопроизводительных вычислительных систем является традиционным для данной области вычислительной математики. Проблематика численного решения дифференциальных уравнений в частных производных является областью интенсивных исследований (см., например, [4,34,55]).

Рассмотрим в качестве учебного примера *проблему численного решения задачи Дирихле для уравнения Пуассона*, определяемую как задачу нахождения функции  $u = u(x, y)$ , удовлетворяющей в области определения  $D$  уравнению

$$\begin{cases} \frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = f(x, y), & (x, y) \in D, \\ u(x, y) = g(x, y), & (x, y) \in D^0, \end{cases}$$

и принимающей значения  $g(x, y)$  на границе  $D^0$  области  $D$  ( $f$  и  $g$  являются функциями, задаваемыми при постановке задачи). Подобная модель может быть использована для описания установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и деформации упругих пластин и др. Данный пример часто используется в качестве учебно-практической задачи при

изложении возможных способов организации эффективных параллельных вычислений (см. [18,21,80]).

Для простоты изложения материала в качестве области задания  $D$  функции  $u(x, y)$  далее будет использоваться единичный квадрат

$$D = \{(x, y) \in D : 0 \leq x, y \leq 1\}.$$

## 11.1. Последовательные методы решения задачи Дирихле

Одним из наиболее распространенных подходов численного решения дифференциальных уравнений является *метод конечных разностей* (*метод сеток*) (см., например, [4,34,80]). Следуя этому подходу, область решения  $D$  представляется в виде дискретного (как правило, равномерного) набора (*сетки*) точек (*узлов*). Так, например, прямоугольная сетка в области  $D$  может быть задана в виде (рис. 12.1)

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih, y_j = jh, 0 \leq i, j \leq N+1, \\ h = 1/(N+1), \end{cases}$$

где величина  $N$  задает количество узлов по каждой из координат области  $D$ .

Обозначим оцениваемую при подобном дискретном представлении аппроксимацию функции  $u(x, y)$  в точках  $(x_i, y_j)$  через  $u_{ij}$ . Тогда, используя *пятиточечный шаблон* (см. рис. 11.1) для вычисления значений производных, мы можем представить уравнение Пуассона в *конечно-разностной форме*

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}.$$

Данное уравнение может быть разрешено относительно  $u_{ij}$

$$u_{ij} = 0.25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Разностное уравнение, записанное в подобной форме, позволяет определять значение  $u_{ij}$  по известным значениям функции  $u(x, y)$  в соседних узлах используемого шаблона. Данный результат служит основой для построения различных *итерационных схем* решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений  $u_{ij}$ , а затем эти значения последовательно уточняются в соответствии с приведенным соотношением. Так, например, *метод Гаусса-Зейделя* для проведения итераций уточнения использует правило

$$u_{ij}^k = 0.25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij}),$$

по которому очередное  $k$ -ое приближение значения  $u_{ij}$  вычисляется по последнему  $k$ -ому приближению значений  $u_{i-1,j}$  и  $u_{i,j-1}$  и предпоследнему  $(k-1)$ -ому приближению значений  $u_{i+1,j}$  и  $u_{i,j+1}$ . Выполнение итераций обычно продолжается до тех пор, пока получаемые в результате итераций изменения значений  $u_{ij}$  не станут меньше некоторой заданной величины (*требуемой точности вычислений*). Сходимость описанной процедуры (получение решения с любой желаемой точностью) является предметом всестороннего математического анализа (см., например, [4,34,80]), здесь же отметим, что последовательность решений, получаемых методом сеток, равномерно сходится к решению задачи Дирихле, а погрешность решения имеет порядок  $h^2$ .

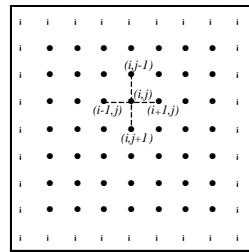


Рис. 11.1. Прямоугольная сетка в области D (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах - сверху вниз)

Рассмотренный алгоритм (метод Гаусса-Зейделя) на псевдокоде, приближенном к алгоритмическому языку C++, может быть представлен в виде:

```
// Алгоритм 11.1
do {
    dmax = 0; // максимальное изменение значений u
    for ( i=1; i<N+1; i++ )
        for ( j=1; j<N+1; j++ ) {
            temp = u[i][j];
            u[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]);
            dm = fabs(temp-u[i][j]);
            if ( dmax < dm ) dmax = dm;
        }
} while ( dmax > eps );
```

Алгоритм 11.1. Последовательный алгоритм Гаусса-Зейделя

(напомним, что значения  $u_{ij}$  при индексах  $i, j = 0, N + 1$  являются граничными, задаются при постановке задачи и не изменяются в ходе вычислений).

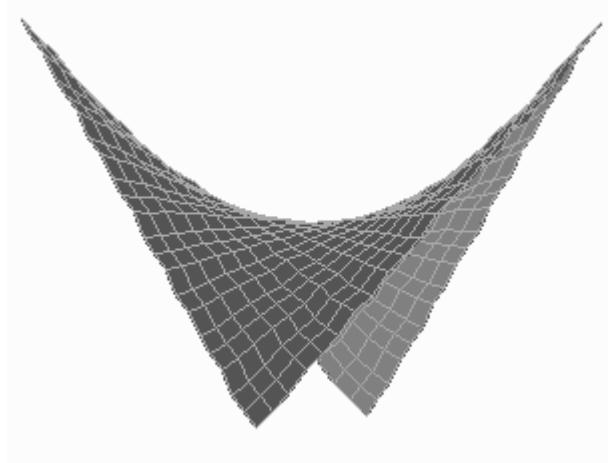


Рис. 11.2. Вид функции  $u(x, y)$  в примере для задачи Дирихле

Для примера на рис. 11.2 приведен вид функции  $u(x, y)$ , полученной для задачи Дирихле при следующих граничных условиях:

$$\begin{cases} f(x, y) = 0, & (x, y) \in D, \\ 100 - 200x, & y = 0, \\ 100 - 200y, & x = 0, \\ -100 + 200x, & y = 1, \\ -100 + 200y, & x = 1, \end{cases}$$

Общее количество итераций метода Гаусса-Зейделя составило 66 при точности решения  $\text{eps} = 0.1$  и  $N = 100$  (в качестве начального приближения величин  $u_{ij}$  использовались значения, сгенерированные датчиком случайных чисел из диапазона [-100,100]).

## 11.2. Организация параллельных вычислений для систем с общей памятью

Как следует из приведенного описания, сеточные методы характеризуются значительной вычислительной трудоемкостью

$$T_1 = kmN^2,$$

где  $N$  есть количество узлов по каждой из координат области  $D$ ,  $m$  - число операций, выполняемых методом для одного узла сетки,  $k$  - количество итераций метода до выполнения условия остановки.

### 11.2.1. Использование OpenMP для организации параллелизма

Рассмотрим возможные способы организации параллельных вычислений для сеточных методов на многопроцессорных вычислительных системах с общей памятью. При изложении материала будем предполагать, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередьность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (*symmetric multiprocessors, SMP*).

Обычный подход при организации вычислений для подобных систем – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки не зависимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное программирование, так и уже имеющиеся языки программирования, расширенные некоторым набором операторов для параллельных вычислений.

Оба указанных подхода приводят к необходимости значительной переработки существующего программного обеспечения, и это в значительной степени затрудняет широкое распространение параллельных вычислений. Как результат, в последнее время активно развивается еще один подход к разработке параллельных программ, когда указания программиста по организации параллельных вычислений добавляются в программу при помощи тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный операторный текст программы остается неизменным, и по нему в случае отсутствия препроцессора компилятор построит исходный последовательный программный код. Препроцессор же, будучи примененным, заменяет директивы параллелизма на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки).

Рассмотренный выше подход является основой технологии *OpenMP* (см., например, [48]), наиболее широко применяемой в настоящее время для организации параллельных вычислений на многопроцессорных системах с общей памятью. В рамках данной

технологии директивы параллелизма используются для выделения в программе *параллельных областей* (*parallel regions*), в которых последовательный исполняемый код может быть разделен на несколько раздельных командных *потоков* (*threads*). Далее эти потоки могут исполняться на разных процессорах вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (*однопоточных*) и параллельных (*многопоточных*) участков программного кода (см. рис. 11.3). Подобный принцип организации параллелизма получил наименование "вилочного" (*fork-join*) или *пульсирующего параллелизма*. Более полная информация по технологии OpenMP может быть получена в главе 5 данного учебного материала, а также в литературе (см., например, [1,48,85]) или в информационных ресурсах сети Интернет; в данном разделе возможности OpenMP будут излагаться в объеме, необходимом для демонстрации возможных способов разработки параллельных программ для рассматриваемого учебного примера решения задачи Дирихле.

### 11.2.2. Проблема синхронизации параллельных вычислений

Первый вариант параллельного алгоритма для метода сеток может быть получен, если разрешить произвольный порядок пересчета значений  $u_{ij}$ . Программа для данного способа вычислений может быть представлена в следующем виде:

```
// Алгоритм 11.2
omp_lock_t dmax_lock;
omp_init_lock (&dmax_lock);
do {
    dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,n,dmax) \
    private(i,j,temp,d)
    for ( i=1; i<N+1; i++ ) {
#pragma omp parallel for shared(u,n,dmax) \
    private(j,temp,d)
        for ( j=1; j<N+1; j++ ) {
            temp = u[i][j];
            u[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]);
            d = fabs(temp-u[i][j]);
            omp_set_lock(&dmax_lock);
            if ( dmax < d ) dmax = d;
            omp_unset_lock(&dmax_lock);
        } // конец вложенной параллельной области
    } // конец внешней параллельной области
} while ( dmax > eps );
```

Алгоритм 11.2. Первый вариант параллельного алгоритма Гаусса-Зейделя

Следует отметить, что программа получена из исходного последовательного кода путем добавления директив и операторов обращения к функциям библиотеки OpenMP (эти дополнительные строки в программе выделены темным шрифтом, обратная наклонная черта в конце директив означает продолжение директив на следующих строках программы).

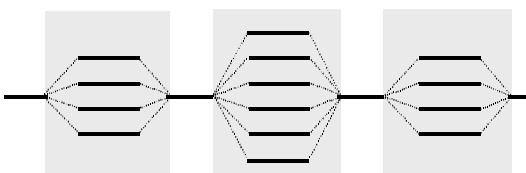


Рис. 11.3. Параллельные области, создаваемые директивами OpenMP

Как следует из текста программы, параллельные области в данном примере задаются директивой *parallel for*, являются вложенными и включают в свой состав операторы цикла *for*. Компилятор, поддерживающий технологию OpenMP, разделяет выполнение итераций цикла между несколькими потоками программы, количество которых обычно совпадает с числом процессоров в вычислительной системе. Параметры директивы *shared* и *private* определяют доступность данных в потоках программы – переменные, описанные как *shared*, являются общими для потоков, для переменных с описанием *private* создаются отдельные копии для каждого потока, которые могут использоваться в потоках независимо друг от друга.

Наличие общих данных обеспечивает возможность взаимодействия потоков. В этом плане разделяемые переменные могут рассматриваться как *общие ресурсы потоков* и, как результат, их использование должно выполняться с соблюдением *правил взаимоисключения* (изменение каким-либо потоком значений общих переменных должно приводить к блокировке доступа к модифицируемым данным для всех остальных потоков). В данном примере таким разделяемым ресурсом является величина *dmax*, доступ потоков к которой регулируется специальной служебной переменной (*семафором*) *dmax\_lock* и функциями *omp\_set\_lock* (разрешение или блокировка доступа) и *omp\_unset\_lock* (снятие запрета на доступ). Подобная организация программы гарантирует единственность доступа потоков для изменения разделяемых данных. Участки программного кода (блоки между обращениями к функциям *omp\_set\_lock* и *omp\_unset\_lock*), для которых обеспечивается взаимоисключение, обычно именуются *критическими секциями*.

Результаты вычислительных экспериментов приведены в табл. 11.1 (здесь и далее для параллельных программ, разработанных с использованием технологии OpenMP, использовался двух процессорный вычислительный узел на базе четырехядерных процессоров Intel Xeon E5320, 1,86 ГГц, 4 Гб RAM под управлением операционной системы Microsoft Windows HPC Server 2008).

**Таблица 11.1.** Результаты вычислительных экспериментов для параллельных вариантов алгоритма Гаусса-Зейделя при использовании 8 потоков ( $p=8$ )

Размер сетки	Последовательный алгоритм 11.1		Параллельный алгоритм 11.2			Параллельный алгоритм 11.3		
	<i>k</i>	<i>t</i>	<i>k</i>	<i>t</i>	<i>S</i>	<i>k</i>	<i>t</i>	<i>S</i>
100	66	0,03	66	0,84	0,04	66	0,01	4,66
200	73	0,13	73	3,72	0,04	73	0,02	5,82
300	74	0,31	74	8,88	0,03	74	0,06	5,18
400	79	0,58	79	17,78	0,03	79	0,10	5,97
500	99	1,14	99	31,89	0,04	99	0,19	6,03
600	92	1,54	92	42,76	0,04	92	0,25	6,16
700	80	1,83	80	50,46	0,04	80	0,30	6,17
800	77	2,32	77	63,77	0,04	77	0,37	6,28
900	109	4,16	109	113,78	0,04	109	0,66	6,28
1000	91	4,30	91	117,55	0,04	91	0,69	6,27
2000	96	18,06	96	499,21	0,04	96	2,92	6,19
3000	93	39,40	93	1092,30	0,04	93	6,74	5,85

(*k* – количество итераций, *t* – время в сек., *S* – ускорение)

Оценим полученный результат. Разработанный параллельный алгоритм является корректным, т.е. обеспечивающим решение поставленной задачи. Использованный при разработке подход обеспечивает достижение практически максимально возможного параллелизма – для выполнения программы может быть задействовано вплоть до  $N^2$  процессоров. Тем не менее, результат не может быть признан удовлетворительным – программа будет работать медленно и вместо ожидаемого ускорения вычислений от использования нескольких процессоров мы получим замедление расчетов (в 25 раз !!!). Основная причина такого положения дел – чрезмерно высокая *синхронизация* параллельных участков программы. В нашем примере каждый параллельный поток после усреднения значений  $u_{ij}$  должен проверить (и возможно, изменить) значение величины  $dmax$ . Разрешение на использование переменной может получить только один поток – все

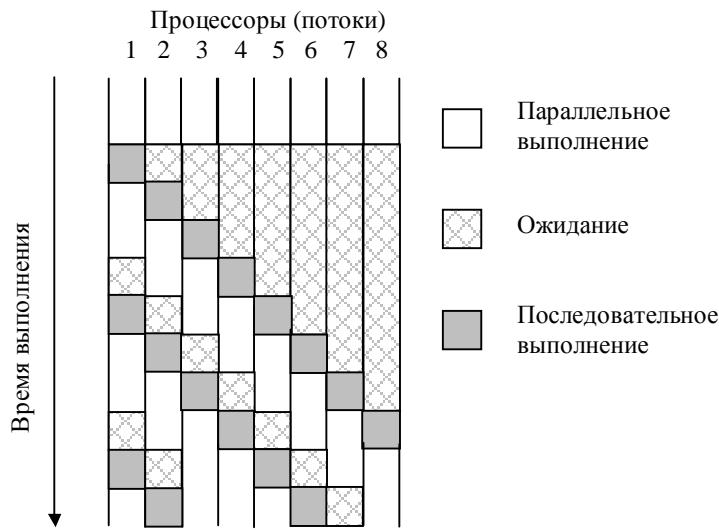


Рис. 11.4. Пример возможной схемы выполнения параллельных потоков при наличии синхронизации (взаимоисключения)

остальные потоки должны быть блокированы. После освобождения общей переменной управление может получить следующий поток и т.д. В результате необходимости синхронизации доступа многопотоковая параллельная программа превращается фактически в последовательно выполняемый код, причем менее эффективный, чем исходный последовательный вариант, т.к. организация синхронизации приводит к дополнительным вычислительным затратам – см. рис. 11.4. Следует обратить внимание, что, несмотря на идеальное распределение вычислительной нагрузки между процессорами, для приведенного на рис. 11.4 соотношения параллельных и последовательных вычислений, в каждый текущий момент времени (после момента первой синхронизации) только не более двух процессоров одновременно выполняют действия, связанные с решением задачи. Подобный эффект вырождения параллелизма из-за интенсивной синхронизации параллельных участков программы обычно именуются *сериализацией (serialization)*.

Как показывают выполненные рассуждения, путь для достижения эффективности параллельных вычислений лежит в уменьшении необходимых моментов синхронизации параллельных участков программы. Так, в нашем примере мы можем ограничиться распараллеливанием только одного внешнего цикла *for*. Кроме того, для снижения количества возможных блокировок применим для оценки максимальной погрешности многоуровневую схему расчета: пусть параллельно выполняемый поток первоначально формирует локальную оценку погрешности  $dm$  только для своих обрабатываемых данных

(одной или нескольких строк сетки), затем при завершении вычислений поток сравнивает свою оценку  $dmax$  с общей оценкой погрешности  $dmax$ .

Новый вариант программы решения задачи Дирихле имеет вид:

```
// Алгоритм 11.3
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
    dmax = 0; // максимальное изменение значений и
#pragma omp parallel for shared(u,n,dmax) \
    private(i,j,temp,d,dm)
    for ( i=1; i<N+1; i++ ) {
        dm = 0;
        for ( j=1; j<N+1; j++ ) {
            temp = u[i][j];
            u[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]);
            d = fabs(temp-u[i][j]);
            if ( dm < d ) dm = d;
        }
        omp_set_lock(dmax_lock);
        if ( dmax < dm ) dmax = dm;
        omp_unset_lock(dmax_lock);
    }
} // конец параллельной области
} while ( dmax > eps );
```

### Алгоритм 11.3. Второй вариант параллельного алгоритма Гаусса-Зейделя

Как результат выполненного изменения схемы вычислений, количество обращений к общей переменной  $dmax$  уменьшается с  $N^2$  до  $N$  раз, что должно приводить к существенному снижению затрат на синхронизацию потоков и уменьшению проявления эффекта сериализации вычислений. Результаты экспериментов с данным вариантом параллельного алгоритма, приведенные в табл. 11.1, показывают существенное изменение ситуации – ускорение в ряде экспериментов приближается к числу вычислительных элементов. Следует также обратить внимание, что улучшение показателей параллельного алгоритма достигнуто при снижении максимально возможного параллелизма (для выполнения программы может использоваться не более  $N$  процессоров).

#### 11.2.3. Возможность неоднозначности вычислений в параллельных программах

Последний рассмотренный вариант организации параллельных вычислений для метода сеток обеспечивает хорошие показатели ускорения выполняемых расчетов – так, в экспериментах данное ускорение достигало величины 6.28 при использовании восьмипроцессорного вычислительного сервера. Вместе с этим необходимо отметить, что разработанная вычислительная схема расчетов имеет важную принципиальную особенность – порождаемая при вычислениях последовательность обработки данных может различаться при разных запусках программы даже при одних и тех же исходных параметрах решаемой задачи. Данный эффект может проявляться в силу изменения каких-либо условий выполнения программы (вычислительной нагрузки, алгоритмов синхронизации потоков и т.п.), что может повлиять на временные соотношения между потоками (см. рис. 11.5). Взаиморасположение потоков по области расчетов может быть различным: одни потоки могут опережать другие и, обратно, часть потоков могут отставать (при этом, характер взаиморасположения может меняться в ходе вычислений). Подобное поведение параллельных участков программы обычно называется *состязанием потоков* (*race condition*) и отражает важный принцип параллельного программирования –

временная динамика выполнения параллельных потоков не должна учитываться при разработке параллельных алгоритмов и программ.

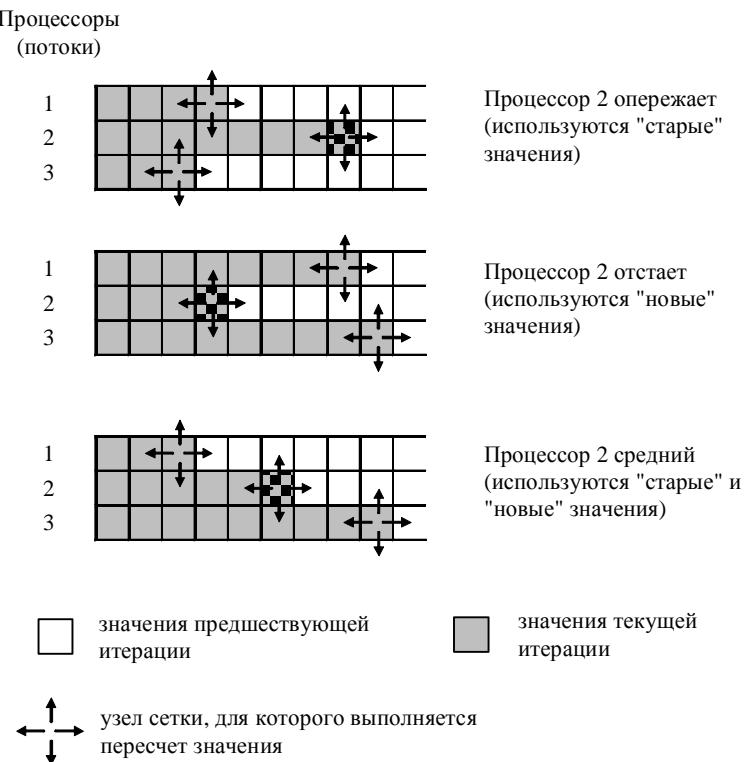


Рис. 11.5. Возможные различные варианты взаиморасположения параллельных потоков (состязание потоков)

В рассматриваемом примере при вычислении нового значения  $u_{ij}$  в зависимости от условий выполнения могут использоваться разные (от предыдущей или текущей итераций) оценки соседних значений по вертикали. Тем самым, количество итераций метода до выполнения условия остановки и, самое главное, конечное решение задачи может различаться при повторных запусках программы. Получаемые оценки величин  $u_{ij}$  будут соответствовать точному решению задачи в пределах задаваемой точности, но, тем не менее, могут быть различными. Использование вычислений такого типа для сеточных алгоритмов получило наименование *метода хаотической релаксации* (*chaotic relaxation*).

#### 11.2.4. Проблема взаимоблокировки

Возможный подход для получения однозначных результатов (уход от состязания потоков) может состоять в ограничении доступа к узлам сетки, которые обрабатываются в параллельных потоках программы. Для этого можно ввести набор семафоров *row\_lock[N]*, который позволит потокам закрывать доступ к "своим" строкам сетки:

```
// поток обрабатывает i строку сетки
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i-1]);
// обработка i строку сетки
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i-1]);
```

Закрыв доступ к своим данным, параллельный поток уже не будет зависеть от динамики выполнения других параллельных участков программы. Результат вычислений потока однозначно определяется значениями данных в момент начала расчетов.

Данный подход позволяет продемонстрировать еще одну проблему, которая может возникать в ходе параллельных вычислений. Эта проблема состоит в том, что при организации доступа к множественным общим переменным может возникать конфликт между параллельными потоками и этот конфликт не может быть разрешен успешно. Так, в приведенном фрагменте программного кода при обработке потоками двух последовательных строк (например, строк 1 и 2) может сложиться ситуация, когда потоки блокируют сначала строки 1 и 2 и только затем переходят к блокировке оставшихся строк (см. рис. 11.6). В этом случае доступ к необходимым строкам не может быть обеспечен ни для одного потока – возникает неразрешимая ситуация, обычно именуемая *тупиком*. Необходимым условием тупика является наличие цикла в графе распределения и запросов ресурсов (см. главу 4 данного учебного материала). В рассматриваемом примере уход от цикла может состоять в строго последовательной схеме блокировки строк потока

```
// поток обрабатывает i строку сетки
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i-1]);
// <обработка i строку сетки>
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i-1]);
```

(следует отметить, что и эта схема блокировки строк может оказаться тупиковой, если рассматривать модифицированную задачу Дирихле, в которой горизонтальные границы являются "склеенными").

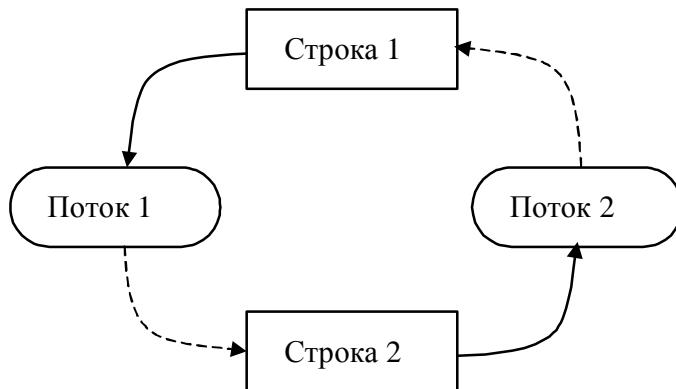


Рис. 11.6. Ситуация тупика при доступе к строкам сетки  
(поток 1 владеет строкой 1 и запрашивает строку 2,  
поток 2 владеет строкой 2 и запрашивает строку 1)

### 11.2.5. Исключение неоднозначности вычислений

Подход, рассмотренный в п. 4, уменьшает эффект состязания потоков, но не гарантирует единственности решения при повторении вычислений. Для достижения однозначности необходимо использование дополнительных вычислительных схем.

Возможный и широко применяемый в практике расчетов способ состоит в разделении места хранения результатов вычислений на предыдущей и текущей итерациях метода сеток. Схема такого подхода может быть представлена в следующем общем виде:

```
// Алгоритм 11.4
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
    dmax = 0; // максимальное изменение значений и
#pragma omp parallel for shared(u,n,dmax) \
```

```

private(i,j,temp,d,dm)
for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
        temp = u[i][j];
        un[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]);
        d = fabs(temp-un[i][j]);
        if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
    if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
}
} // конец параллельной области
for ( i=1; i<N+1; i++ ) // обновление данных
    for ( j=1; j<N+1; j++ )
        u[i][j] = un[i][j];
} while ( dmax > eps );

```

Алгоритм 11.4. Параллельная реализация сеточного метода Гаусса-Якоби

Как следует из приведенного алгоритма, результаты предыдущей итерации запоминаются в массиве  $u$ , новые вычисления значения запоминаются в дополнительном массиве  $un$ . Как результат, независимо от порядка выполнения вычислений для проведения расчетов всегда используются значения величин  $u_{ij}$  от предыдущей итерации метода. Такая схема реализации сеточных алгоритмов обычно называется *методом Гаусса-Якоби*. Этот метод гарантирует однозначность результатов независимо от способа распараллеливания, но требует использования большого дополнительного объема памяти и обладает меньшей (по сравнению с алгоритмом Гаусса-Зейделя) скоростью сходимости. Результаты расчетов с последовательным и параллельным вариантами метода приведены в табл. 11.2.

**Таблица 11.2.** Результаты вычислительных экспериментов для алгоритма Гаусса-Якоби при использовании 8 потоков ( $p=8$ )

Размер сетки	Последовательный метод Гаусса-Якоби		Параллельный метод 11.4		
	<i>k</i>	<i>t</i>	<i>k</i>	<i>t</i>	<i>S</i>
100	6360	3,2812	6360	2,4166	1,36
200	21098	42,1318	21098	25,6422	1,64
300	42578	190,1464	42578	110,2716	1,72
400	35833	287,4144	35833	160,9858	1,79
500	101605	1278,1038	101605	709,1393	1,80
600	68225	1235,8253	68225	690,1085	1,79
700	191552	4722,7376	191552	2593,8825	1,82
800	113719	3662,0482	113719	1965,9076	1,86
900	216404	8819,8531	216404	4519,3607	1,95
1000	299932	15091,5502	299932	7494,8095	2,01
2000	565537	113823,533	565537	47877,9222	2,38
3000	888744	402466,966	888744	164051,634	2,45

(*k* – количество итераций, *t* – время в сек., *S* – ускорение)

Иной возможный подход для устранения взаимозависимости параллельных потоков состоит в применении *схемы чередования обработки четных и нечетных строк (red/black row alternation scheme)*, когда выполнение итерации метода сеток подразделяется на два последовательных этапа, на первом из которых обрабатываются строки только с четными номерами, а затем на втором этапе - строки с нечетными номерами (см. рис. 11.7). Данная схема может быть обобщена на применение одновременно и к строкам, и к столбцам (*шахматное разбиение*) области расчетов.

Рассмотренная схема чередования строк не требует по сравнению с методом Якоби какой-либо дополнительной памяти и обеспечивает однозначность решения при многократных запусках программы. Но следует заметить, что оба рассмотренных в данном пункте подхода могут получать результаты, не совпадающие с решением задачи Дирихле, найденном при помощи последовательного алгоритма. Кроме того, эти вычислительные схемы имеют меньшую область и худшую скорость сходимости, чем исходный вариант метода Гаусса-Зейделя.

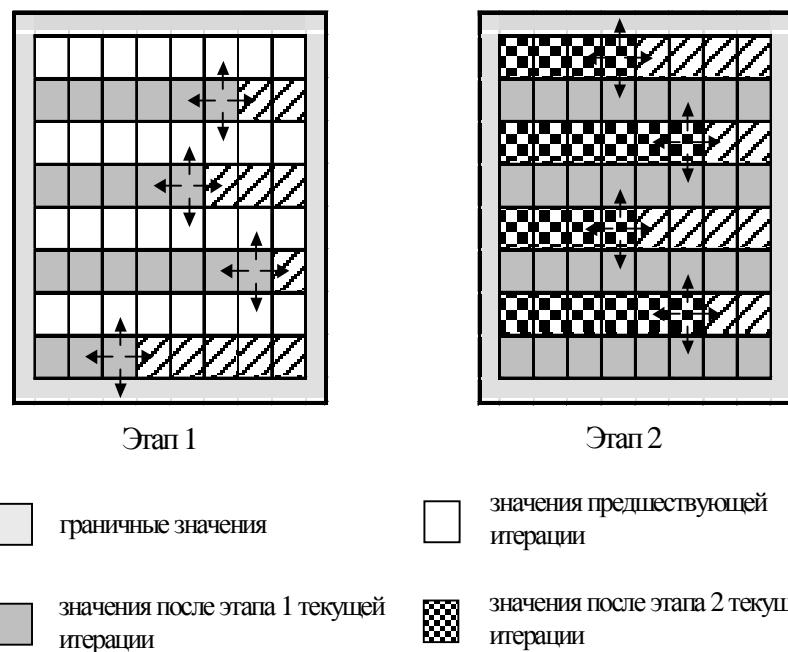


Рис. 11.7. Схема чередования обработки четных и нечетных строк

### 11.2.6. Волновые схемы параллельных вычислений

Рассмотрим теперь возможность построения параллельного алгоритма, который выполнял бы только те вычислительные действия, что и последовательный метод (может быть только в некотором ином порядке) и, как результат, обеспечивал бы получение точно таких же решений исходной вычислительной задачи. Как уже было отмечено выше, в последовательном алгоритме каждое очередное  $k$ -ое приближение значения  $u_{ij}$  вычисляется по последнему  $k$ -ому приближению значений  $u_{i-1,j}$  и  $u_{i,j-1}$  и предпоследнему  $(k-1)$ -ому приближению значений  $u_{i+1,j}$  и  $u_{i,j+1}$ . Как результат, при требовании совпадения результатов вычислений последовательных и параллельных вычислительных схем в начале каждой итерации метода только одно значение  $u_{11}$  может быть пересчитано (возможности для распараллеливания нет). Но далее после пересчета  $u_{11}$  вычисления могут выполняться уже в двух узлах сетки  $u_{12}$  и  $u_{21}$  (в этих узлах выполняются условия последовательной схемы), затем после пересчета узлов  $u_{12}$  и  $u_{21}$  - в узлах  $u_{13}$ ,  $u_{22}$  и  $u_{31}$  и т.д. Обобщая сказанное, можно увидеть, что выполнение итерации метода сеток можно

разбить на последовательность шагов, на каждом из которых к вычислениям окажутся подготовленными узлы вспомогательной диагонали сетки с номером, определяемом номером этапа – см. рис. 11.8. Получаемая в результате вычислительная схема получила наименование *волны* или *фронт вычислений*, а алгоритмы, получаемые на ее основе, – *методами волновой обработки данных* (*wavefront or hyperplane methods*). Следует отметить, что в нашем случае размер волны (степень возможного параллелизма) динамически изменяется в ходе вычислений – волна нарастает до своего пика, а затем затухает при приближении к правому нижнему узлу сетки.

**Таблица 11.3.** Результаты экспериментов для параллельных вариантов алгоритма Гаусса-Зейделя с волновой схемой расчета при использовании 8 потоков ( $p=8$ )

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 11.1)		Параллельный алгоритм 11.5			Параллельный алгоритм 11.6		
	<i>k</i>	<i>t</i>	<i>k</i>	<i>t</i>	<i>S</i>	<i>k</i>	<i>t</i>	<i>S</i>
100	66	0,03	66	0,14	0,23	66	0,11	0,28
200	73	0,13	73	0,13	1,05	73	0,09	1,50
300	74	0,31	74	0,23	1,33	74	0,20	1,51
400	79	0,58	79	0,37	1,59	79	0,22	2,64
500	99	1,14	99	0,82	1,39	99	0,57	1,99
600	92	1,54	92	0,79	1,96	92	0,63	2,45
700	80	1,83	80	0,88	2,08	80	0,79	2,31
800	77	2,32	77	1,07	2,17	77	0,96	2,41
900	109	4,16	109	2,14	1,95	109	2,03	2,05
1000	91	4,30	91	1,89	2,28	91	1,47	2,92
2000	96	18,06	96	5,90	3,06	96	4,66	3,87
3000	93	39,40	93	10,89	3,62	93	8,72	4,52

(*k* – количество итераций, *t* – время в сек., *S* – ускорение)

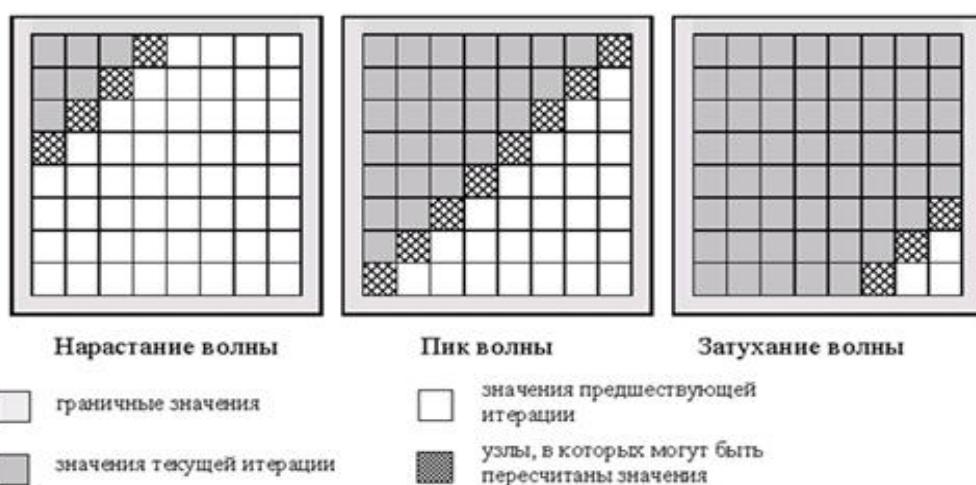


Рис. 11.8. Движение фронта волны вычислений

Возможная схема параллельного метода, основанного на эффекте волны вычислений, может быть представлена в следующей форме:

```

// Алгоритм 11.5
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
    dmax = 0; // максимальное изменение значений и
    // нарастание волны (nx - размер волны)
    for ( nx=1; nx<N+1; nx++ ) {
        dm[nx] = 0;
#pragma omp parallel for shared(u,nx,dm) \
    private(i,j,temp,d)
        for ( i=1; i<nx+1; i++ ) {
            j      = nx + 1 - i;
            temp = u[i][j];
            u[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]);
            d = fabs(temp-u[i][j]);
            if ( dm[i] < d ) dm[i] = d;
        } // конец параллельной области
    }
    // затухание волны
    for ( nx=N-1; nx>0; nx-- ) {
#pragma omp parallel for shared(u,nx,dm) \
    private(i,j,temp,d)
        for ( i=N-nx+1; i<N+1; i++ ) {
            j      = 2*N - nx - I + 1;
            temp = u[i][j];
            u[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]);
            d = fabs(temp-u[i][j]);
            if ( dm[i] < d ) dm[i] = d;
        } // конец параллельной области
    }
#pragma omp parallel for shared(n,dm,dmax) \
    private(i)
    for ( i=1; i<nx+1; i++ ) {
        omp_set_lock(dmax_lock);
        if ( dmax < dm[i] ) dmax = dm[i];
        omp_unset_lock(dmax_lock);
    } // конец параллельной области
} while ( dmax > eps );

```

### Алгоритм 11.5. Параллельный алгоритм, реализующий волновую схему вычислений

При разработке алгоритма, реализующего волновую схему вычислений, оценку погрешности решения можно осуществлять для каждой строки в отдельности (массив значений  $dm$ ). Этот массив является общим для всех выполняемых потоков, однако, синхронизации доступа к элементам не требуется, так как потоки используют всегда разные элементы массива (фронт волны вычислений содержит только по одному узлу строк сетки).

После обработки всех элементов волны в составе массива  $dm$  находится максимальная погрешность выполненной итерации вычислений. Однако именно эта последняя часть расчетов может оказаться наиболее неэффективной из-за высоких дополнительных затрат на синхронизацию. Улучшение ситуации, как и ранее, может быть достигнуто за счет увеличения размера последовательных участков и сокращения, тем самым, количества необходимых взаимодействий параллельных участков вычислений. Возможный вариант реализации такого подхода может состоять в следующем:

```

chunk = 200; // размер последовательного участка
#pragma omp parallel for shared(n,dm,dmax) \

```

```

private(i,d)
for ( i=1; i<nx+1; i+=chunk ) {
    d = 0;
    for ( j=i; j<i+chunk; j++ )
        if ( d < dm[j] ) d = dm[j];
    omp_set_lock(dmax_lock);
    if ( dmax < d ) dmax = d;
    omp_unset_lock(dmax_lock);
} // конец параллельной области

```

Подобный прием укрупнения последовательных участков вычислений для снижения затрат на синхронизацию именуется *фрагментированием* (*chunking*). Результаты экспериментов для данного варианта параллельных вычислений приведены в табл. 11.3.

Следует обратить внимание еще на один момент при анализе эффективности разработанного параллельного алгоритма. Фронт волны вычислений плохо соответствует правилам использования кэша - быстродействующей дополнительной памяти компьютера, используемой для хранения копии наиболее часто используемых областей оперативной памяти. Использование кэша может существенно повысить (в десятки раз) быстродействие вычислений. Размещение данных в кэше может происходить или предварительно (при использовании тех или иных алгоритмов предсказания потребности в данных) или в момент извлечения значений из основной оперативной памяти. При этом подкачка данных в кэш осуществляется не одиночными значениями, а небольшими группами – *строками кэша* (*cache line*). Загрузка значений в строку кэша осуществляется из последовательных элементов памяти; типовые размеры строки кэша обычно равны 32, 64, 128, 256 байтам (дополнительная информация по организации памяти может быть получена, например, в (см., например, [35]). Как результат, эффект наличия кэша будет наблюдаться, если выполняемые вычисления используют одни и те же данные многократно (*локальность обработки данных*) и осуществляют доступ к элементам памяти с последовательно возрастающими адресами (*последовательность доступа*).

В рассматриваемом нами алгоритме размещение данных в памяти осуществляется по строкам, а фронт волны вычислений располагается по диагонали сетки, и это приводит к низкой эффективности использования кэша. Возможный способ улучшения ситуации – опять же укрупнение вычислительных операций и рассмотрение в качестве распределяемых между процессорами действий процедуру обработки некоторой прямоугольной подобласти (*блока*) сетки расчетов - см. рис. 11.9.

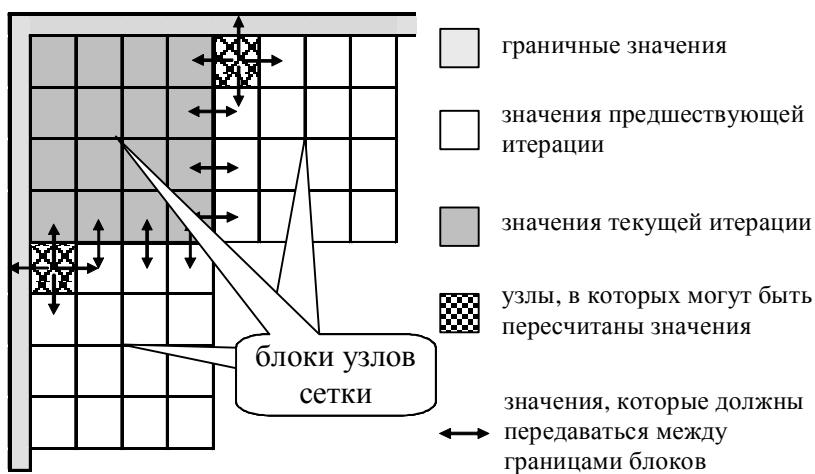


Рис. 11.9. Блочное представление сетки области расчетов

Порождаемый на основе такого подхода метод вычислений в самом общем виде может быть описан следующим образом (блоки образуют в области расчётов прямоугольную решётку размера  $NB \times NB$ ):

```
// Алгоритм 11.6
// NB количество блоков
do {
    // нарастание волны (размер волны равен nx+1)
    for ( nx=0; nx<NB; nx++ ) {
#pragma omp parallel for shared(nx) private(i,j)
        for ( i=0; i<nx+1; i++ ) {
            j = nx - i;
            // <обработка блока с координатами (i,j)>
        } // конец параллельной области
    }
    // затухание волны
    for ( nx=NB-2; nx>-1; nx-- ) {
#pragma omp parallel for shared(nx) private(i,j)
        for ( i=0; i<nx+1; i++ ) {
            j = 2*(NB-1) - nx - i;
            // <обработка блока с координатами (i,j)>
        } // конец параллельной области
    }
    // <определение погрешности вычислений>
} while ( dmax > eps );
```

Алгоритм 11.6. Блочный подход к методу волновой обработки данных

Вычисления в предлагаемом алгоритме происходят в соответствии с волновой схемой обработки данных – вначале вычисления выполняются только в левом верхнем блоке с координатами (0,0), далее для обработки становятся доступными блоки с координатами (0,1) и (1,0) и т.д. – см. результаты экспериментов в табл. 11.3.

Блочный подход к методу волновой обработки данных существенным образом меняет состояние дел – обработку узлов можно организовать построчно, доступ к данным осуществляется последовательно по элементам памяти, перемещенные в кэш значения используются многократно. Кроме того, поскольку обработка блоков будет выполняться на разных процессорах и блоки не пересекаются по данным, при таком подходе будут отсутствовать и накладные расходы для обеспечения однозначности (*когерентности*) кэшей разных процессоров.

Наилучшие показатели использования кэша будут достигаться, если в кэше будет достаточно места для размещения не менее трех строк блока (при обработке строки блока используются данные трех строк блока одновременно). Тем самым, исходя из размера кэша, можно определить рекомендуемый максимально-возможный размер блока. Так, например, при кэше 8 Кб и 8-байтовых значениях данных этот размер составит приближенно 300 (8Кб/3/8). Можно определить и минимально-допустимый размер блока из условия совпадения размеров строк кэша и блока. Так, при размере строки кэша 256 байт и 8-байтовых значениях данных размер блока должен быть кратен 32.

Последнее замечание необходимо сделать о взаимодействии граничных узлов блоков. Учитывая граничное взаимодействие, соседние блоки целесообразно обрабатывать на одних и тех же процессорах. В противном случае можно попытаться так определить размеры блоков, чтобы объем пересылаемых между процессорами граничных данных был минимален. Так, при размере строки кэша в 256 байт, 8-байтовых значениях данных и размере блока 64x64 объем пересылаемых данных 132 строки кэша, при размере блока 128x32 – всего 72 строки. Такая оптимизация имеет наиболее принципиальное значение при медленных операциях пересылки данных между кэшами процессоров, т.е. для систем с неоднородным доступом к памяти (*nonuniform memory access - NUMA*).

### 11.2.7. Балансировка вычислительной нагрузки процессоров

Как уже отмечалось ранее, вычислительная нагрузка при волновой обработке данных изменяется динамически в ходе вычислений. Данный момент следует учитывать при распределении вычислительной нагрузки между процессорами. Так, например, при фронте волны из 5 блоков и при использовании 4 процессоров обработка волны потребует двух параллельных итераций, во время второй из которых будет задействован только один процессор, а все остальные процессоры будут простаивать, дожидаясь завершения вычислений. Достигнутое ускорение расчетов в этом случае окажется равным 2.5 вместо потенциально возможного значения 4. Подобное снижение эффективности использования процессоров становится менее заметным при большой длине волны, размер которой может регулироваться размером блока. Как общий результат, можно отметить, что размер блока определяет *степень разбиения (granularity)* вычислений для распараллеливания и является параметром, подбирая значения которого, можно управлять эффективностью параллельных вычислений.

Для обеспечения равномерности (*балансировки*) загрузки процессоров можно задействовать еще один подход, широко используемый для организации параллельных вычислений. Этот подход состоит в том, что все готовые к выполнению в системе вычислительные действия организуются в виде *очереди заданий*. В ходе вычислений освободившийся процессор может запросить для себя работу из этой очереди; появляющиеся по мере обработки данных дополнительные задания пополняют задания очереди. Такая схема балансировки вычислительной нагрузки между процессорами является простой, наглядной и эффективной. Это позволяет говорить об использовании очереди заданий как об *общей модели организации параллельных вычислений* для систем с общей памятью.

Рассмотренная схема балансировки может быть задействована и для рассматриваемого примера. В самом деле, в ходе обработки фронта текущей волны происходит постепенное формирование блоков следующей волны вычислений. Эти блоки могут быть задействованы для обработки при нехватке достаточной вычислительной нагрузки для процессоров.

Общая схема вычислений с использованием очереди заданий может быть представлена в следующем виде:

```
// Алгоритм 11.7
// <инициализация служебных данных>
// <загрузка в очередь указателя на начальный блок>
// взять блок из очереди (если очередь не пуста)
while ( (pBlock=GetBlock()) != NULL ) {
    // <обработка блока>
    // отметка готовности соседних блоков
    omp_set_lock(pBlock->pNext.Lock); // сосед справа
    pBlock->pNext.Count++;
    if ( pBlock->pNext.Count == 2 )
        PutBlock(pBlock->pNext);
    omp_unset_lock(pBlock->pNext.Lock);
    omp_set_lock(pBlock->pDown.Lock); // сосед снизу
    pBlock->pDown.Count++;
    if ( pBlock->pDown.Count == 2 )
        PutBlock(pBlock->pDown);
    omp_unset_lock(pBlock->pDown.Lock);
} // завершение вычислений, т.к. очередь пуста
```

Алгоритм 11.7. Общая схема вычислений с использованием очереди

Для описания имеющихся в задаче блоков узлов сетки в алгоритме используется структура со следующим набором параметров:

- *Lock* – семафор, синхронизирующий доступ к описанию блока,
- *pNext* – указатель на соседний справа блок,
- *pDown* – указатель на соседний снизу блок,
- *Count* – счетчик готовности блока к вычислениям (количество готовых границ блока).

Операции для выборки из очереди и вставки в очередь указателя на готовый к обработке блок узлов сетки обеспечивают соответственно функции *GetBlock* и *PutBlock*.

Как следует из приведенной схемы, процессор извлекает блок для обработки из очереди, выполняет необходимые вычисления для блока и отмечает готовность своих границ для соседних справа и снизу блоков. Если при этом оказывается, что у соседних блоков являются подготовленными обе границы, процессор передает эти блоки для запоминания в очередь заданий.

Использование очереди заданий позволяет решить практически все оставшиеся вопросы организации параллельных вычислений для систем с общей памятью. Развитие рассмотренного подхода может предусматривать уточнение правил выделения заданий из очереди для согласования с состояниями процессоров (близкие блоки целесообразно обрабатывать на одних и тех же процессорах), расширение числа имеющихся очередей заданий и т.п. Дополнительная информация по этим вопросам может быть получена, например, в (см., например, [85, 100]).

### **11.3. Организация параллельных вычислений для систем с распределенной памятью**

Использование процессоров с распределенной памятью является другим общим способом построения многопроцессорных вычислительных систем. Актуальность таких систем становится все более высокой в последнее время в связи с широким развитием высокопроизводительных кластерных вычислительных систем (см. [10]).

Многие проблемы параллельного программирования (состязание вычислений, туники, сериализация) являются общими для систем с общей и распределенной памятью. Момент, который отличает параллельные вычисления с распределенной памятью, состоит в том, что взаимодействие параллельных участков программы на разных процессорах может быть обеспечено только при помощи *передачи сообщений* (*message passing*).

Следует отметить, что процессор с распределенной памятью является, как правило, более сложным вычислительным устройством, чем процессор в многопроцессорной системе с общей памятью. Для учета этих различий в дальнейшем процессор с распределенной памятью будет именоваться как *вычислительный сервер* (сервером может быть, в частности, многопроцессорная система с общей памятью). Эксперименты проводились на вычислительной системе из двух серверов, каждый из которых имел по два четырехядерных процессоров Intel Xeon E5320, 1.86 ГГц, 4 Гб RAM. Для передачи данных между серверами применялась сеть Gigabit Ethernet. В качестве операционной системы использовался Microsoft Windows HPC Server 2008. Разработка программ проводилась в среде Microsoft Visual Studio 2008, для компиляции использовался Intel C++ Compiler 10.0 for Windows.

#### **11.3.1. Разделение данных**

Первая проблема, которую приходится решать при организации параллельных вычислений на системах с распределенной памяти, обычно состоит в выборе способа разделения обрабатываемых данных между вычислительными серверами. Успешность такого разделения определяется достигнутой степенью локализации вычислений на

серверах (в силу больших временных задержек при передаче сообщений интенсивность взаимодействия серверов должна быть минимальной).

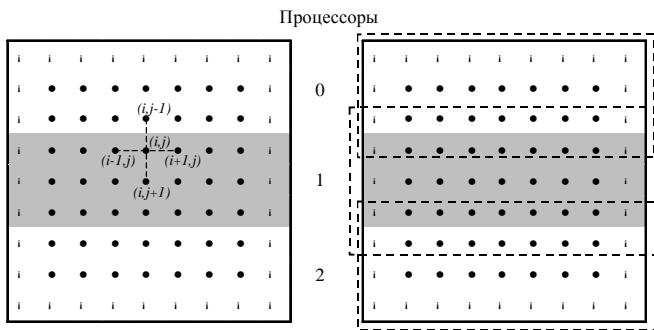


Рис. 11.10. Ленточное разделение области расчетов между процессорами (кружки представляют граничные узлы сетки)

В рассматриваемой учебной задаче по решению задачи Дирихле возможны два различных способа разделения данных – *одномерная* или *ленточная* схема (см. рис. 11.10) или *двухмерное* или *блочное* разбиение (см. рис. 11.9) вычислительной сетки. Дальнейшее изложение учебного материала будет проводиться на примере первого подхода; блочная схема будет рассмотрена позднее в более кратком виде.

При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы (не уменьшая общности, далее будем рассматривать только горизонтальные полосы). Число полос определяется количеством процессоров, размер полос обычно является одинаковым, узлы горизонтальных границ (первая и последняя строки) включаются в первую и последнюю полосы соответственно. Полосы для обработки распределяются между процессорами.

Основной момент при организации вычислений с подобным разделением данных состоит в том, что на процессор, выполняющий обработку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки (получаемые в результате расширенные полосы показаны на рис. 11.10 справа пунктирными рамками). Продублированные граничные строки полос используются только при проведении расчетов, пересчет же этих строк происходит в полосах своего исходного месторасположения. Тем самым дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток.

### 11.3.2. Обмен информацией между процессорами

Параллельный вариант метода сеток при ленточном разделении данных состоит в обработке полос на всех имеющихся серверах одновременно в соответствии со следующей схемой работы:

```
// Алгоритм 11.8
// схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
    // <обмен граничных строк полос с соседями>
    // <обработка полосы>
    // <вычисление общей погрешности вычислений dmax>
} while ( dmax > eps ); // eps - точность решения
```

Алгоритм 11.8. Параллельный алгоритм, реализующий метод сеток при ленточном разделении данных

Для конкретизации представленных в алгоритме действий введем обозначения:

- $ProcNum$  – номер процессора, на котором выполняются описываемые действия,
- $PrevProc$ ,  $NextProc$  – номера соседних процессоров, содержащих предшествующую и следующую полосы,
- $NP$  – количество процессоров,
- $M$  – количество строк в полосе (без учета продублированных граничных строк),
- $N$  – количество внутренних узлов в строке сетки (т.е. всего в строке  $N+2$  узла).

Для нумерации строк полосы будем использовать нумерацию, при которой строки 0 и  $M+1$  есть продублированные из соседних полос граничные строки, а строки собственной полосы процессора имеют номера от 1 до  $M$ .

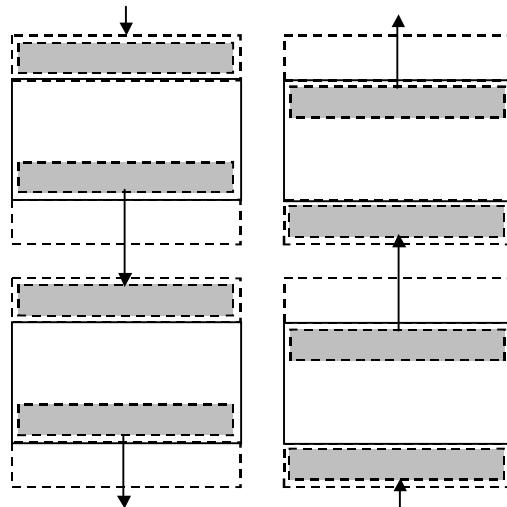


Рис. 11.11. Схема передачи граничных строк между соседними процессорами

Процедура обмена граничных строк между соседними процессорами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора (см. рис. 11.11). Вторая часть передачи строк выполняется в обратном направлении: процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров.

Выполнение подобных операций передачи данных в общем виде может быть представлено следующим образом (для краткости рассмотрим только первую часть процедуры обмена):

```
// передача нижней граничной строки следующему процессору
// и прием передаваемой строки от предыдущего процессора
if ( ProcNum != NP-1 )Send(u[M][*],N+2,NextProc);
if ( ProcNum != 0 )Receive(u[0][*],N+2,PrevProc);
```

(для записи процедур приема-передачи используется близкий к стандарту MPI (см., например, [62]) формат, где первый и второй параметры представляют пересылаемые данные и их объем, а третий параметр определяет адресат (для операции *Send*) или источник (для операции *Receive*) пересылки данных).

Для передачи данных могут быть задействованы два различных механизма. При первом из них выполнение программ, инициировавших операцию передачи, приостанавливается до полного завершения всех действий по пересылке данных (т.е. до момента получения процессором-адресатом всех передаваемых ему данных). Операции приема-передачи, реализуемые подобным образом, обычно называются *синхронными* или *блокирующими*. Иной подход – *асинхронная* или *неблокирующая* передача – может

состоять в том, что операции приема-передачи только инициируют процесс пересылки и на этом завершают свое выполнение. В результате программы, не дожидаясь завершения длительных коммуникационных операций, могут продолжать свои вычислительные действия, проверяя по мере необходимости готовность передаваемых данных. Оба эти варианта операций передачи широко используются при организации параллельных вычислений и имеют свои достоинства и свои недостатки. Синхронные процедуры передачи, как правило, более просты для использования и более надежны; неблокирующие операции могут позволить совместить процессы передачи данных и вычислений, но обычно приводят к повышению сложности программирования. С учетом всех последующих примеров для организации пересылки данных будут использоваться операции приема-передачи блокирующего типа.

Приведенная выше последовательность блокирующих операций приема-передачи данных (вначале *Send*, затем *Receive*) приводит к строго последовательной схеме выполнения процесса пересылок строк, т.к. все процессоры одновременно обращаются к операции *Send* и переходят в режим ожидания. Первым процессором, который окажется готовым к приему пересылаемых данных, окажется сервер с номером *NP-1*. В результате процессор *NP-2* выполнит операцию передачи своей граничной строки и перейдет к приему строки от процессора *NP-3* и т.д. Общее количество повторений таких операций равно *NP-1*. Аналогично происходит выполнение и второй части процедуры пересылки граничных строк перед началом обработки строк (см. рис. 11.11).

Последовательный характер рассмотренных операций пересылок данных определяется выбранным способом очередности выполнения. Изменим этот порядок очередности при помощи чередования приема и передачи для процессоров с четными и нечетными номерами:

```
// передача нижней граничной строки следующему процессору
// и прием передаваемой строки от предыдущего процессора
if ( ProcNum % 2 == 1 ) { // нечетный процессор
    if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
    if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
}
else { // процессор с четным номером
    if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
    if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
}
```

Данный прием позволяет выполнить все необходимые операции передачи всего за два последовательных шага. На первом шаге все процессоры с нечетными номерами отправляют данные, а процессоры с четными номерами осуществляют прием этих данных. На втором шаге роли процессоров меняются – четные процессоры выполняют *Send*, нечетные процессоры исполняют операцию приема *Receive*.

Рассмотренные последовательности операций приема-передачи для взаимодействия соседних процессоров широко используются в практике параллельных вычислений. Как результат, во многих базовых библиотеках параллельных программ имеются процедуры для поддержки подобных действий. Так, в стандарте MPI (см., например, [62]) предусмотрена операция *Sendrecv*, с использованием которой предыдущий фрагмент программного кода может быть записан более кратко:

```
// передача нижней граничной строки следующему процессору
// и прием передаваемой строки от предыдущего процессора
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
```

Реализация подобной объединенной функции *Sendrecv* обычно осуществляется таким образом, чтобы обеспечить и корректную работу на крайних процессорах, когда не нужно выполнять одну из операций передачи или приема, и организацию чередования процедур

передачи на процессорах для ухода от тупиковых ситуаций, и возможности параллельного выполнения всех необходимых пересылок данных.

### 11.3.3. Коллективные операции обмена информацией

Для завершения круга вопросов, связанных с параллельной реализацией метода сеток на системах с распределенной памятью, осталось рассмотреть способы вычисления общей для всех процессоров погрешности вычислений. Возможный очевидный подход состоит в передаче всех локальных оценок погрешности, полученных на отдельных полосах сетки, на один какой-либо процессор, вычисления на нем максимального значения и последующей рассылки полученного значения всем процессорам системы. Однако такая схема является крайне неэффективной – количество необходимых операций передачи данных определяется числом процессоров и выполнение этих операций может происходить только в последовательном режиме. Между тем, как показывает анализ требуемых коммуникационных действий, выполнение операций сборки и рассылки данных может быть реализовано с использованием рассмотренной в главе 2 *каскадной схемы* обработки данных. На самом деле, получение максимального значения локальных погрешностей, вычисленных на каждом процессоре, может быть обеспечено, например, путем предварительного нахождения максимальных значений для отдельных пар процессоров (данные вычисления могут выполняться параллельно), затем может быть снова осуществлен попарный поиск максимума среди полученных результатов и т.д. Всего, как полагается по каскадной схеме, необходимо выполнить  $\log_2 NP$  параллельных итераций для получения конечного значения ( $NP$  – количество процессоров).

Учитывая большую эффективность каскадной схемы для выполнения коллективных операций передачи данных, большинство базовых библиотек параллельных программ содержит процедуры для поддержки подобных действий. Так, в стандарте MPI (см., например, [62]) предусмотрены операции:

- *Reduce(dm,dmax,op,proc)* – процедура сборки на процессоре *proc* итогового результата *dmax* среди локальных на каждом процессоре значений *dm* с применением операции *op*,
- *Broadcast(dmax,proc)* – процедура рассылки с процессора *proc* значения *dmax* всем имеющимся процессорам системы.

С учетом перечисленных процедур общая схема вычислений на каждом процессоре может быть представлена в следующем виде:

```
// Алгоритм 11.8 – уточненный вариант
// схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
    // обмен граничных строк полос с соседями
    Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
    Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
    // <обработка полосы с оценкой погрешности dm>
    // вычисление общей погрешности вычислений dmax
    Reduce(dm,dmax,MAX,0);
    Broadcast(dmax,0);
} while ( dmax > eps ); // eps – точность решения
```

(в приведенном алгоритме переменная *dm* представляет локальную погрешность вычислений на отдельном процессоре, параметр *MAX* задает операцию поиска максимального значения для операции сборки). Следует отметить, что в составе MPI имеется процедура *Allreduce*, которая совмещает действия редукции и рассылки данных. Результаты экспериментов для данного варианта параллельных вычислений для метода Гаусса-Зейделя приведены в табл. 11.4.

#### 11.3.4. Организация волны вычислений

Представленные в пп. 1-3 алгоритмы определяют общую схему параллельных вычислений для метода сеток в многопроцессорных системах с распределенной памятью. Далее эта схема может быть конкретизирована реализацией практически всех вариантов методов, рассмотренных для систем с общей памятью (использование дополнительной памяти для схемы Гаусса-Якоби, чередование обработки полос и т.п.). Проработка таких вариантов не привносит каких-либо новых эффектов с точки зрения параллельных вычислений, и их разбор может использоваться как темы заданий для самостоятельных упражнений.

**Таблица 11.4.** Результаты экспериментов для систем с распределенной памятью, ленточная схема разделения данных ( $p=16$ )

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 11.1)		Параллельный алгоритм 11.8			Параллельный алгоритм с волновой схемой расчета (см. 11.3.4)		
	<i>k</i>	<i>t</i>	<i>k</i>	<i>t</i>	<i>S</i>	<i>k</i>	<i>t</i>	<i>S</i>
100	66	0,03	66	0,05	0,62	66	0,01	4,66
200	73	0,13	73	0,08	1,69	73	0,02	5,82
300	74	0,31	74	0,12	2,59	74	0,06	5,18
400	79	0,58	79	0,20	2,96	79	0,1	5,97
500	99	1,14	99	0,35	3,26	99	0,19	6,03
600	92	1,54	92	0,45	3,40	92	0,25	6,16
700	80	1,83	80	0,51	3,56	80	0,3	6,17
800	77	2,32	77	0,64	3,65	77	0,37	6,28
900	109	4,16	109	1,13	3,68	109	0,66	6,28
1000	91	4,3	91	1,15	3,74	91	0,69	6,27
2000	96	18,06	96	4,82	3,75	96	2,92	6,19
3000	93	39,4	93	10,10	3,90	93	6,74	5,85

(*k* – количество итераций, *t* – время в сек., *S* – ускорение)

В завершение рассмотрим возможность организации параллельных вычислений, при которых обеспечивалось бы нахождение таких же решений задачи Дирихле, что и при использовании исходного последовательного метода Гаусса-Зейделя. Как отмечалось ранее, такой результат может быть получен за счет организации волновой схемы расчетов. Для образования волны вычислений представим логически каждую полосу узлов области расчетов в виде набора блоков (размер блоков можно положить, в частности, равным ширине полосы) и организуем обработку полос поблочно в последовательном порядке (см. рис. 11.12). Тогда для полного повторения действий последовательного алгоритма вычисления могут быть начаты только для первого блока первой полосы узлов; после того, как этот блок будет обработан, для вычислений будут готовы уже два блока – блок 2 первой полосы и блок 1 второй полосы (для обработки блока полосы 2 необходимо передать граничную строку узлов первого блока полосы 1). После обработки указанных блоков к вычислениям будут готовы уже 3 блока, и мы получаем знакомый уже процесс волновой обработки данных (результаты экспериментов см. в табл. 11.4).

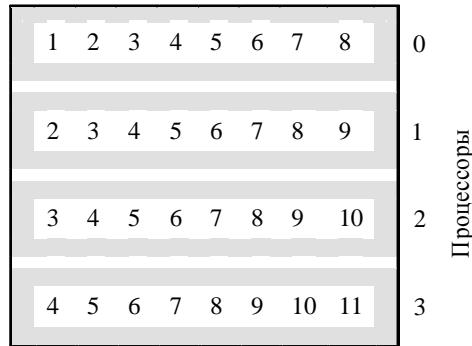


Рис. 11.12. Организация волны вычислений при ленточной схеме разделения данных

Интересный момент при организации подобной схемы параллельных вычислений может состоять в попытке совмещения операций пересылки граничных строк и действий по обработке блоков данных.

### 11.3.5. Блочная схема разделения данных

Ленточная схема разделения данных может быть естественным образом обобщена на блочный способ представления сетки области расчетов (см. рис. 11.9). При этом столь радикальное изменение способа разбиения сетки практически не потребует каких-либо существенных корректировок рассмотренной схемы параллельных вычислений. Основной новый момент при блочном представлении данных состоит в увеличении количества граничных строк на каждом процессоре (для блока их количество становится равным 4), что приводит, соответственно, к большему числу операций передачи данных при обмене граничных строк. Сравнивая затраты на организацию передачи граничных строк, можно отметить, что при ленточной схеме для каждого процессора выполняется 4 операции приема-передачи данных, в каждой из которых пересылаются  $(N+2)$  значения. Для блочного же способа происходит 8 операций пересылки и объем каждого сообщения равен  $(N/\sqrt{NP} + 2)$  ( $N$  – количество внутренних узлов сетки,  $NP$  – число процессоров, размер всех блоков предполагается одинаковым). Тем самым, блочная схема представления области расчетов становится оправданной при большом количестве узлов сетки области расчетов, когда увеличение количества коммуникационных операций приводит к снижению затрат на пересылку данных в силу сокращения размеров передаваемых сообщений. Результаты экспериментов при блочной схеме разделения данных приведены в табл. 11.5.

**Таблица 11.5.** Результаты экспериментов для систем с распределенной памятью, блочная схема разделения данных ( $p=16$ )

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 11.1)		Параллельный алгоритм с блочной схемой расчета (см. п. 11.3.5)			Параллельный алгоритм 11.9		
	<i>k</i>	<i>t</i>	<i>k</i>	<i>t</i>	<i>S</i>	<i>k</i>	<i>t</i>	<i>S</i>
100	66	0,03	66	0,40	0,08	66	0,24	0,13
200	73	0,13	73	0,40	0,32	73	0,40	0,32
300	74	0,31	74	0,35	0,89	74	0,54	0,57
400	79	0,58	79	0,31	1,88	79	0,51	1,14
500	99	1,14	99	0,49	2,33	99	0,92	1,23

600	92	1,54	92	0,38	4,04	92	0,74	2,09
700	80	1,83	80	0,37	4,92	80	0,73	2,52
800	77	2,32	77	0,37	6,29	77	0,79	2,93
900	109	4,16	109	1,00	4,17	109	2,13	1,96
1000	91	4,3	91	0,66	6,49	91	1,50	2,86
2000	96	18,06	96	2,59	6,98	96	6,36	2,84
3000	93	39,4	93	4,78	8,24	93	10,44	3,77

( $k$  – количество итераций,  $t$  – время в сек.,  $S$  – ускорение)

При блочном представлении сетки может быть реализован также и волновой метод выполнения расчетов (см. рис. 11.13). Пусть процессоры образуют прямоугольную решетку размером  $NB \times NB$  ( $NB = \sqrt{NP}$ ) и процессоры пронумерованы от 0 слева направо по строкам решетки.

Общая схема параллельных вычислений в этом случае имеет вид:

```
// Алгоритм 11.9
// схема Гаусса-Зейделя, блочное разделение данных
// действия, выполняемые на каждом процессоре
do {
    // получение граничных узлов
    if ( ProcNum / NB != 0 ) { // строка не нулевая
        // получение данных от верхнего процессора
        Receive(u[0][*],M+2,TopProc); // верхняя строка
        Receive(dmax,1,TopProc); // погрешность
    }
    if ( ProcNum % NB != 0 ) { // столбец не нулевой
        // получение данных от левого процессора
        Receive(u[*][0],M+2,LeftProc); // левый столбец
        Receive(dm,1,LeftProc); // погрешность
        If ( dm > dmax ) dmax = dm;
    }
    // <обработка блока с оценкой погрешности dmax>
    // пересылка граничных узлов
    if ( ProcNum / NB != NB-1 ) { // строка решетки не
        // последняя
        // пересылка данных нижнему процессору
        Send(u[M+1][*],M+2,DownProc); // нижняя строка
        Send(dmax,1,DownProc); // погрешность
    }
    if ( ProcNum % NB != NB-1 ) { // столбец решетки
        // не последний
        // пересылка данных правому процессору
        Send(u[*][M+1],M+2,RightProc); // правый столбец
        Send(dmax,1, RightProc); // погрешность
    }
    // синхронизация и рассылка погрешности dmax
    Barrier();
    Broadcast(dmax,NP-1);
} while ( dmax > eps ); // eps - точность решения
```

### Алгоритм 11.9. Блочная схема разделения данных

(в приведенном алгоритме функция *Barrier()* представляет операцию коллективной синхронизации, которая завершает свое выполнение только в тот момент, когда все процессоры осуществляют вызов этой процедуры).

Следует обратить внимание, что при реализации алгоритма нужно обеспечить, чтобы в начальный момент времени все процессоры (кроме процессора с нулевым номером) оказались в состоянии передачи своих граничных узлов (верхней строки и левого

столбца). Вычисления должен начинать процессор с левым верхним блоком, после завершения обработки которого обновленные значения правого столбца и нижней строки блока необходимо переправить правому и нижнему процессорам решетки соответственно. Данные действия обеспечат снятие блокировки процессоров второй диагонали процессорной решётки (ситуация слева на рис. 11.13) и т.д.

Анализ эффективности организации волновых вычислений в системах с распределенной памятью (см. табл. 11.5) показывает значительное снижение полезной вычислительной нагрузки для процессоров, которые занимаются обработкой данных только в моменты, когда их блоки попадают во фронт волны вычислений. При этом балансировка (перераспределение) нагрузки является крайне затруднительной, поскольку связана с пересылкой между процессорами блоков данных большого объема. Возможный интересный способ улучшения ситуации состоит в организации *множественной волны вычислений*, в соответствии с которой процессоры после отработки волны текущей итерации расчетов могут приступить к выполнению волны следующей итерации метода сеток. Так, например, процессор 0 (см. рис. 11.13), передав после обработки своего блока граничные данные и запустив, тем самым, вычисления на процессорах 1 и 4, оказывается готовым к исполнению следующей итерации метода Гаусса-Зейделя. После обработки блоков первой (процессорах 1 и 4) и второй (процессор 0) волн, к вычислениям окажутся готовыми следующие группы процессоров (для первой волны - процессоры 2, 5 и 8, для второй волны - процессоры 1 и 4). Кроме того, процессор 0 опять окажется готовым к запуску очередной волны обработки данных. После выполнения  $NB$  подобных шагов в обработке будет находиться одновременно  $NB$  итераций и все процессоры окажутся задействованными. Подобная схема организации расчетов позволяет рассматривать имеющуюся процессорную решетку как *вычислительный конвейер* поэтапного выполнения итераций метода сеток. Останов конвейера может осуществляться, как и ранее, по максимальной погрешности вычислений (проверку условия остановки следует начинать только при достижении полной загрузки конвейера после запуска  $NB$  итераций расчетов). Необходимо отметить также, что получаемое после выполнения условия остановки решение задачи Дирихле будет содержать значения узлов сетки от разных итераций метода и не будет, тем самым, совпадать с решением, получаемым при помощи исходного последовательного алгоритма.

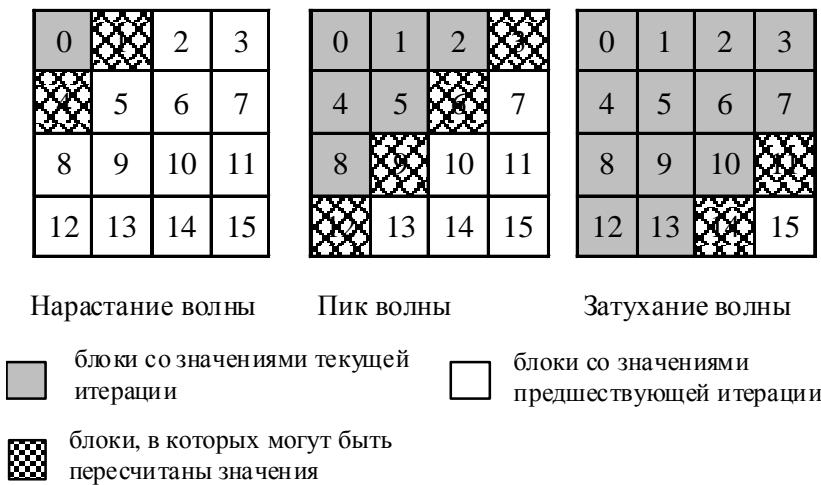


Рис. 11.13. Организация волны вычислений при блочной схеме разделения данных

### 11.3.6. Оценка трудоемкости операций передачи данных

Время выполнения коммуникационных операций значительно превышает длительность вычислительных команд. Оценка трудоемкости операций приема-передачи может быть осуществлена с использованием двух основных характеристик сети передачи: *латентности* (*latency*), определяющей время подготовки данных к передаче по сети, и *пропускной способности* сети (*bandwidth*), задающей объем передаваемых по сети за 1 сек. данных – более полное изложение вопроса содержится в [10].

Пропускная способность наиболее распространенной на данный момент сети Fast Ethernet – 100 Мбит/с, для более современной сети Gigabit Ethernet – 1000 Мбит/с. В то же время, скорость передачи данных в системах с общей памятью обычно составляет сотни и тысячи миллионов байт в секунду. Тем самым, использование систем с распределенной памятью приводит к снижению скорости передачи данных не менее чем в 100 раз.

Еще хуже дело обстоит с латентностью. Для сети Fast Ethernet эта характеристика имеет значений порядка 150 мкс, для сети Gigabit Ethernet – около 100 мкс. Для современных компьютеров с тактовой частотой выше 2 ГГц/с различие в производительности достигает не менее, чем 10000-100000 раз. При указанных характеристиках вычислительной системы для достижения 90% эффективности в рассматриваемом примере решения задачи Дирихле (т.е. чтобы в ходе расчетов обработка данных занимала не менее 90% времени от общей длительности вычислений и только 10% времени тратилось бы на операции передачи данных) размер блоков вычислительной сетки должен быть не менее  $N=7500$  узлов по вертикали и горизонтали (объем вычислений в блоке составляет  $5N^2$  операций с плавающей запятой).

Как результат, можно заключить, что эффективность параллельных вычислений при использовании распределенной памяти определяется в основном интенсивностью и видом выполняемых коммуникационных операций при взаимодействии процессоров. Необходимый при этом анализ параллельных методов и программ может быть выполнен значительно быстрее за счет выделения типовых операций передачи данных. Так, например, в рассматриваемой учебной задаче решения задачи Дирихле практически все пересылки значений сводятся к стандартным коммуникационным действиям, имеющим адекватную поддержку в стандарте MPI (см. рис. 11.14):

- рассылка количества узлов сетки всем процессорам – типовая операция передачи данных от одного процессора всем процессорам сети (функция *MPI\_Bcast*);
- рассылка полос или блоков узлов сетки всем процессорам – типовая операция передачи разных данных от одного процессора всем процессорам сети (функция *MPI\_Scatter*);
- обмен граничных строк или столбцов сетки между соседними процессорами – типовая операция передачи данных между соседними процессорами сети (функция *MPI\_Sendrecv*);

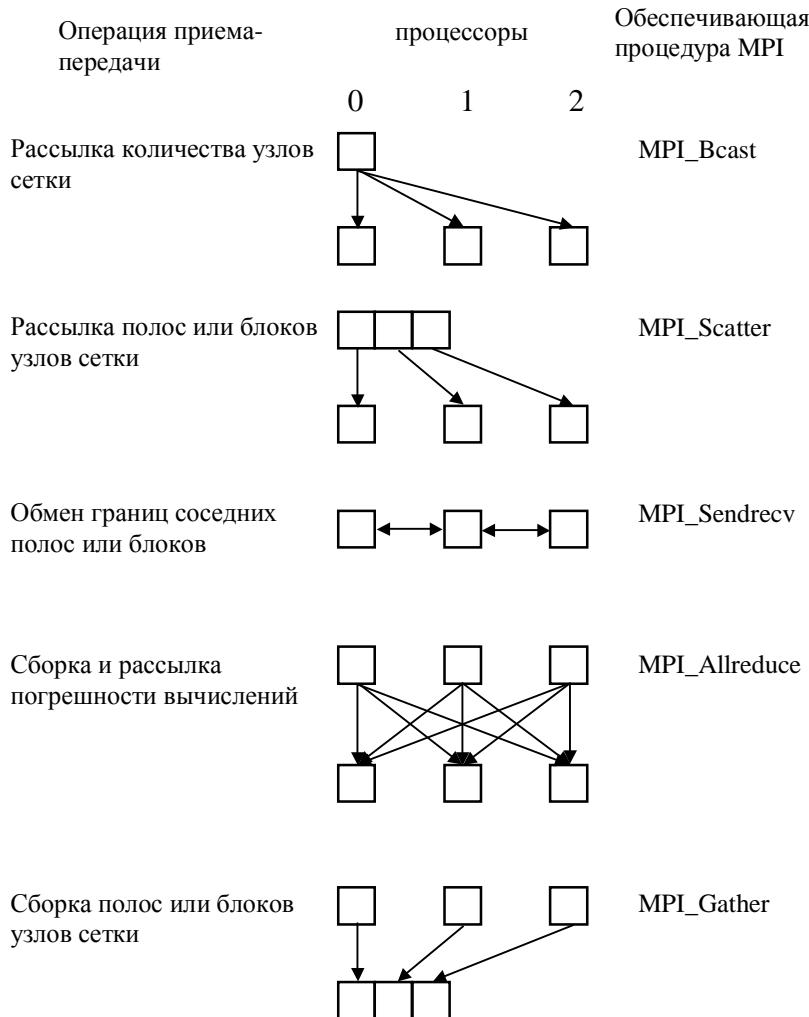


Рис. 11.14. Операции передачи данных при выполнении метода сеток в системе с распределенной памятью

- сборка и рассылка погрешности вычислений всем процессорам – типовая операция передачи данных от всех процессоров всем процессорам сети (функция *MPI\_Allreduce*);
- сборка на одном процессоре решения задачи (всех полос или блоков сетки) – типовая операция передачи данных от всех процессоров сети одному процессору (функция *MPI\_Gather*).

## 11.4. Краткий обзор главы

В главе описываются модели, построенные на основе дифференциальных уравнений в частных производных. В качестве учебного примера рассматривается проблема численного решения задачи Дирихле для уравнения Пуассона. Наиболее распространенный подход численного решения дифференциальных уравнений является метод конечных разностей, при этом объем выполняемых вычислений является значительным. В разделе рассматриваются возможные способы организации параллельных вычислений для сеточных методов на многопроцессорных вычислительных системах с общей и распределенной памятью.

При изложении организации параллельных вычислений для систем с общей памятью основное внимание уделяется технологии OpenMP, также приводятся проблемы, возникающие при применении этой технологии и решения данных проблем. Стоить отметить решение проблем взаимоблокировки через семафоры и исключение

неоднозначности вычислений путем применения схемы чередования обработки четных и нечетных строк.

При изложении организации параллельных вычислений для систем с распределенной памятью основное внимание уделяется разделению данных и обмену информацией между процессорами. Рассматриваются различные механизмы приема - передачи данных, такие как синхронные и асинхронные. В завершении приводятся оценки трудоемкости операций передачи данных с использованием двух характеристик: латентности и пропускной способности.

## 11.5. Обзор литературы

Дополнительная информация по исследованию проблематики численного решения дифференциальных уравнений в частных производных и по методу конечных разностей может быть получена в [4,34], полезная информация содержится также в [55].

Рассмотрение вопроса по организации памяти и использованию кэша приводится в [35].

Информация по вопросам, связанным с использованием очередей заданий, а именно, правила выделения заданий из очереди для согласования с состояниями процессоров и расширения числа имеющихся очередей заданий может быть получена в [85,100].

## 11.6. Контрольные вопросы

1. Как определяется задача Дирихле для уравнения Пуассона?
2. Как метод конечных разностей применяется для решения задачи Дирихле?
3. Какие способы определяют организацию параллельных вычислений для сеточных методов на многопроцессорных вычислительных системах с общей памятью?
4. В чем состоит проблема синхронизации параллельных вычислений?
5. Как характеризуется поведение параллельных участков программы, которое называется состязанием потоков (race condition)?
6. В чем состоит проблема взаимоблокировки?
7. Какой метод гарантирует однозначность результатов сеточных методов независимо от способа распараллеливания, но требует использования большого дополнительного объема памяти?
8. Как изменяется объем вычислений при применении методов волновой обработки данных?
9. Что такое фрагментирование?
10. Как повысить эффективность методов волновой обработки данных?
11. Как очередь заданий позволяет балансировать нагрузку процессорам?
12. Какие проблемы приходится решать при организации параллельных вычислений на системах с распределенной памятью?
13. Какие механизмы могут быть задействованы для передачи данных?
14. Каким образом организация множественной волны вычислений позволяет повысить эффективность волновых вычислений в системах с распределенной памятью?

## 11.7. Задачи и упражнения

1. Выполните реализацию первого и второго вариантов параллельного алгоритма Гаусса-Зейделя. Сравните время выполнения выполненных реализаций.

2. Выполните реализации параллельного алгоритма на основе волновой схемы вычислений и параллельного алгоритма, в котором реализуется блочный подход к методу волновой обработки данных. Сравните время выполнения реализаций.

3. Выполните реализацию очереди заданий параллельных вычислений для систем с общей памятью. При реализации необходимо обеспечить уточнение правил выделения заданий из очереди для согласования с состояниями процессоров, когда близкие блоки обрабатываются на одних и тех же процессорах.

<b>ГЛАВА 12. МНОГОЭКСТРЕМАЛЬНАЯ ОПТИМИЗАЦИЯ .....</b>	<b>1</b>
12.1. Введение .....	1
12.2. Постановка задачи .....	1
12.3. Методы решения задач многоэкстремальной оптимизации .....	3
12.4. Решение одномерных задач .....	4
12.4.1. Индексный метод учета ограничений.....	4
12.4.2. Схема алгоритма.....	8
12.4.3. Программная реализация .....	10
12.4.4. Результаты численных экспериментов .....	14
12.5. Редукция размерности задачи .....	15
12.5.1. Использование отображений Пеано .....	15
12.5.2. Программная реализация .....	16
12.5.3. Результаты численных экспериментов .....	19
12.5.4. Способ построения развертки .....	20
12.6. Использование множественных отображений .....	22
12.6.1. Основная схема.....	22
12.6.2. Программная реализация .....	23
12.6.3. Способ построения множественных отображений.....	23
12.7. Параллельный индексный метод .....	25
12.7.1. Организация параллельных вычислений.....	25
12.7.2. Схема алгоритма.....	26
12.7.3. Результаты численных экспериментов .....	28

## Глава 12.

### Многоэкстремальная оптимизация

#### **12.1. Введение**

Во всех областях своей целенаправленной деятельности перед человеком возникает проблема выбора наилучшего решения из множества всех возможных. Примерами могут служить экономика (управление экономическими объектами), техника (выбор оптимальной конструкции). К числу наиболее распространенных моделей рационального выбора относятся математические задачи оптимизации (максимизации или минимизации) некоторого функционала при ограничениях типа неравенств. При этом выполнение ограничений для некоторого вектора параметров, определяющего решение, интерпретируется как допустимость этого решения, т.е. как возможность его реализации при имеющихся ресурсах. Теорию и методы отыскания минимумов функций многих переменных при наличии дополнительных ограничений на эти переменные обычно рассматривают как отдельную дисциплину – математическое программирование (см., например, [6,14,52,60]).

Следует отметить, что, если в главах 6-11 рассматриваемые задачи имели, скорее всего, учебный характер и не требовали для своего изучения каких-либо значительных усилий, то исследуемая в данном разделе проблема глобальной оптимизации предполагает более глубокое погружение в тематику и ее успешное освоение требует значительно большего времени. Изложение проблематики глобального поиска осуществляется в достаточно кратком виде; для получения более полной информации следует обратиться к основной литературе в данной области – см., например, [13,28,97].

#### **12.2. Постановка задачи**

В общем виде задачу математического программирования можно сформулировать следующим образом. Пусть  $j(y)$ ,  $g_j(y) \leq 0$ ,  $1 \leq j \leq m$ , есть действительные функции,

определенные на множестве  $D$   $N$ -мерного евклидова пространства  $R^N$ , и пусть точка  $y^*$  удовлетворяет условию

$$j(y^*) = \min\{j(y) : y \in D, g_j(y) \leq 0, 1 \leq j \leq m\}. \quad (12.1)$$

Точка  $y^*$  из (12.1) обычно называется *глобально-оптимальной точкой* или *глобально-оптимальным решением*. При этом функцию  $j(y)$  называют *функцией цели*, или *целевой функцией*, а функции  $g_j(y) \leq 0, 1 \leq j \leq m$ , - *ограничениями задачи*.

Область  $D$  называют *областью поиска* и обычно описывают как некоторый гиперинтервал из  $N$ -мерного евклидова пространства

$$D = \{y \in R^N : a_i \leq y_i \leq b_i, 1 \leq i \leq n\},$$

где  $a, b \in R^N$  есть заданные векторы. Точки из области поиска, удовлетворяющие всем ограничениям, называются *допустимыми точками* или *допустимыми решениями*. Множество

$$Q = \{y : y \in D, g_j(y) \leq 0, 1 \leq j \leq m\} \quad (12.2)$$

всех таких точек называют *допустимой областью*.

Важный в прикладном отношении подкласс задач вида (12.1) характеризуется тем, что все функционалы, входящие в определение задачи, заданы некоторыми (программно реализуемыми) алгоритмами вычисления значений  $j(y)$ ,  $g_j(y)$ ,  $1 \leq j \leq m$ , в точках области поиска  $D$ . При этом *решение задачи* (12.1) сводится к построению оценки  $y_* \in Q$ , отвечающей некоторому понятию близости к точке  $y^*$  (например, чтобы  $\|y^* - y_*\| \leq e$  или  $|j(y^*) - j(y_*)| \leq e$ , где  $e > 0$  есть заданная точность) на основе некоторого числа  $k$  значений функционалов задачи, вычисленных в точках области  $D$ .

В задачах многоэкстремальной оптимизации возможность достоверной оценки глобального оптимума принципиально основана на наличии *априорной информации* о функции, позволяющей связать возможные значения минимизируемой функции с известными значениями в точках осуществленных поисковых итераций. Весьма часто такая априорная информация о задаче (12.1) представляется в виде предположения, что целевая функция  $j$  (в дальнейшем обозначаемая также  $g_{m+1}$ ) и левые части ограничений  $g_j, 1 \leq j \leq m$ , удовлетворяют *условию Липшица* с соответствующими константами  $L_j, 1 \leq j \leq m+1$ , а именно

$$|g_j(y_1) - g_j(y_2)| \leq L_j \|y_1 - y_2\|, \quad 1 \leq j \leq m+1, \quad y_1, y_2 \in D. \quad (12.3)$$

В общем случае все эти функции могут быть *многоэкстремальными*.

*Пример 12.1.* Рассмотрим задачу минимизации функции

$$\begin{aligned} j(y_1, y_2) = & -1.5y_1^2 \exp\{1 - y_1^2 - 20.25(y_1 - y_2)^2\} - \\ & - [0.5(y_1 - 1)(y_2 - 1)]^4 \exp\{2 - [0.5(y_1 - 1)]^4 - (y_2 - 1)^4\} \end{aligned}$$

в области поиска  $0 \leq y_1 \leq 4, -1 \leq y_2 \leq 3$ , при ограничениях

$$g_1(y_1, y_2) = 0.01[(y_1 - 2.2)^2 + (y_2 - 1.2)^2 - 2.25] \leq 0$$

$$g_2(y_1, y_2) = 100[1 - (y_1 - 2)^2 / 1.44 - (0.5y_2)^2] \leq 0$$

$$g_3(y_1, y_2) = 10[y_2 - 1.5 - 1.5 \sin(6.283(y_1 - 1.75))] \leq 0$$

Допустимые по первому ограничению точки образуют круг с границей  $g_1(y_1, y_2)=0$ . Допустимые по второму ограничению точки находятся во внешности эллипса  $g_2(y_1, y_2)=0$ . Точки, допустимые по третьему ограничению, находятся ниже синусоиды  $g_3(y_1, y_2)=0$ . Следовательно, допустимая область является неодносвязной и состоит из трех невыпуклых подобластей (на рис. 12.1 допустимая область выделена цветом). Глобальный минимум  $j(y_1^*, y_2^*) = -1.489$  достигается в точке  $(y_1^*, y_2^*)=(0.942, 0.944)$ .

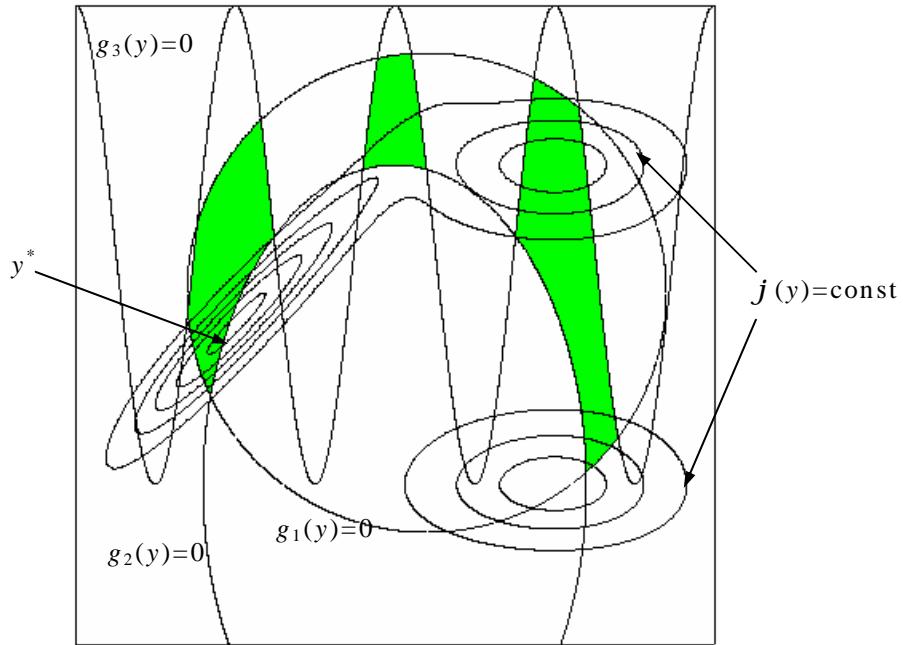


Рис. 12.1. Задача глобальной оптимизации

### 12.3. Методы решения задач многоэкстремальной оптимизации

Важно отметить, что в задачах многоэкстремальной оптимизации глобальный экстремум является интегральной характеристикой задачи, его определение, вообще говоря, связано с построением покрытия области поиска точками вычислений функции. Поэтому на сложность решения задач рассматриваемого класса решающее влияние оказывает размерность: они подвержены “проклятию размерности”, состоящему в экспоненциальном росте вычислительных затрат при увеличении размерности. Использование простейших способов решения (таких, как перебор по равномерной сетке) является чрезмерно затратным. Требуется применение более экономных методов, которые порождают в области поиска существенно неравномерную сетку, более плотную в окрестности решения задачи. Рассмотрению вопросов, возникающих при построении подобного рода методов, посвящены, например, работы [28,52,66,74,82-83,97,101].

Фундаментальные результаты в этом направлении были получены Нижегородской школой глобальной оптимизации (коллектив исследователей под руководством Р.Г. Стронгина, Нижегородский государственный университет): разработан информационно-статистический подход к построению алгоритмов многоэкстремальной оптимизации ([28,97]). Общая информация о подходе содержится в указанных работах, здесь же кратко поясним следующее. Минимизируемая функция рассматривается как реализация некоторого случайного процесса, и решающие правила алгоритма конструируются таким образом, что очередная итерация проводится в точке глобального минимума математического ожидания значений функции.

В рамках информационно-статистического подхода был разработан эффективный способ учета ограничений в задачах условной оптимизации, получивший название *индексного метода* (см. [80,106]). Его характерной чертой является раздельный учет каждого из ограничений задачи, штрафные функции не используются. В соответствии с

правилами индексного метода каждая итерация, называемая *испытанием* в соответствующей точке области поиска, включает последовательную проверку выполнимости ограничений задачи в этой точке, а обнаружение первого нарушенного ограничения прерывает испытание и инициирует переход к точке следующей итерации.

Следует отметить, что регулярные поисковые методы решения многомерных оптимизационных задач (т.е. методы, отличные от стохастических процедур типа Монте-Карло) как правило (явно или неявно) сводят многомерную задачу к системе одномерных подзадач. Локальные методы, например, осуществляют такое сведение путем построения последовательности направлений спуска, вдоль которых осуществляется одномерная минимизация, что порождает траекторию, ведущую из начальной точки в окрестность решения (см., например, [51,60,66]). В многоэкстремальных задачах схема локального спуска, вообще говоря, не приводит к решению. Редукция размерности при решении таких задач может основываться на некоторых фундаментальных свойствах многомерных функций и многомерных пространств.

Один из подходов сводит решение многомерной задачи к решению серии вложенных одномерных подзадач (т.н. *многошаговая схема*). Описание параллельного алгоритма решения многомерных задач многоэкстремальной оптимизации, основанного на многошаговой схеме редукции размерности, приводится, например, в работе [91].

Другой способ редукции размерности, речь о котором пойдет ниже, использует отображение многомерной области поиска на одномерный интервал с помощью *кривых Пеано*. Итальянским математиком Пеано было установлено (см. [79]), что может быть построено однозначное непрерывное отображение  $y(x)=(y_1(x), y_2(x))$  отрезка  $[0,1]$  на единичный квадрат:

$$\{y \in R^2: -2^{-1} \leq y_1, y_2 \leq 2^{-1}\} = \{y(x): 0 \leq x \leq 1\}.$$

Этот результат был обобщен на многомерный случай, т.е. было доказано существование кривых  $y(x)$ , заданных непрерывными координатными функциями  $y_i(x)$ ,  $x \in [0,1]$ ,  $1 \leq i \leq N$ , однозначно отображающих отрезок  $[0,1]$  на  $N$ -мерный гиперкуб  $D$ , т.е.

$$D = \{y \in R^N: a_i \leq y_i \leq b_i, 1 \leq i \leq N\} = \{y(x): 0 \leq x \leq 1\}.$$

Такие кривые, называемые также *развертками Пеано*, позволяют свести многомерную задачу условной минимизации в области  $D$  к одномерной задаче условной минимизации на отрезке  $[0,1]$

$$j(y(x^*)) = \min\{j(y(x)): x \in [0,1], g_j(y(x)) \leq 0, 1 \leq j \leq m\}. \quad (12.4)$$

Таким образом, рассмотренная схема сведения многомерной многоэкстремальной задачи условной оптимизации к эквивалентной ей одномерной задаче позволяет применить для ее решения эффективные одномерные методы поиска. Для более простого освоения материала изложим сначала индексную схему учета ограничений для одномерных задач.

## 12.4. Решение одномерных задач

### 12.4.1. Индексный метод учета ограничений

Рассмотрим одномерную задачу условной глобальной оптимизации вида

$$j(x^*) = \min\{j(x): x \in [a, b], g_j(x) \leq 0, 1 \leq j \leq m\} \quad (12.5)$$

в предположении, что целевая функция  $j$  (в дальнейшем обозначаемая также  $g_{m+1}$ , т.е.  $g_{m+1}(x) \equiv j(x)$ ) и левые части ограничений  $g_j, 1 \leq j \leq m$ , являются определенными на

отрезке  $[a, b]$  липшицевыми функциями с соответствующими константами  $L_j, 1 \leq j \leq m+1$ . В общем случае все эти функции могут быть многоэкстремальными.

*Пример 12.2.* Рассмотрим задачу вида (12.5) при  $x \in [0.6, 2.2]$ ,  $m=2$ ,

$$j(x) = \cos(18x-3)\sin(10x-7)+1.5,$$

$$g_1(x) = \exp(-x/2)\sin(6x-1.5),$$

$$g_2(x) = |x|\sin(2\pi x-0.5).$$

Точное решение задачи  $x^* = 2.0795$ ,  $j(x^*) = 0.565$ . Графики функций  $g_1(x)$ ,  $g_2(x)$ ,  $j(x)$  представлены на рис. 12.2.

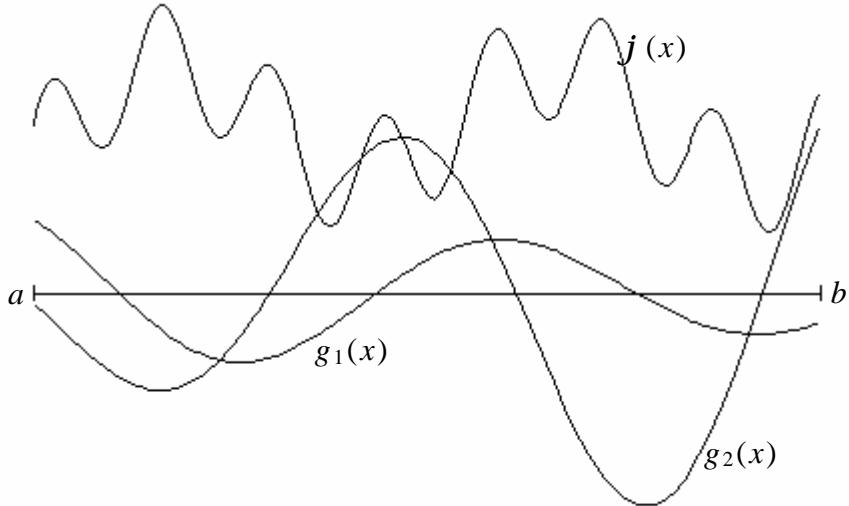


Рис. 12.2. Одномерная задача глобальной оптимизации

Задача (12.5) может быть рассмотрена в постановке, когда каждая функция  $g_j$ ,  $1 \leq j \leq m+1$ , определена и вычислима лишь в соответствующей подобласти  $Q_j \subset [a, b]$ , где

$$Q_1 = [a, b], Q_{j+1} = \{x \in Q_j : g_j(x) \leq 0\}, 1 \leq j \leq m. \quad (12.6)$$

Так, например, в задачах оптимального проектирования некоторые характеристики технической системы оказываются неопределенными, если не выполнены представленные частью ограничений задачи (12.5) условия функционирования системы.

С учетом условий (12.6), исходная задача (12.5) может быть представлена в виде

$$j(x^*) = \min\{g_{m+1}(x) : x \in Q_{m+1}\}. \quad (12.7)$$

Для целей дальнейшего изложения введем классификацию точек  $x$  из области поиска  $[a, b]$  с помощью *индекса*  $n=n(x)$ , где  $n-1$  есть число ограничений, которые выполняются в этой точке. Указанный индекс  $n$  определяется условиями

$$g_j(x) \leq 0, 1 \leq j \leq n-1, g_n(x) > 0, \quad (12.8)$$

где последнее неравенство несущественно, если  $n=m+1$ , и удовлетворяет неравенствам

$$1 \leq n = n(x) \leq m+1.$$

На рис. 12.3 приведена задача из примера 12.2 в предположении частичной вычислимости функций. Изображены дуги функций ограничений  $g_1(x)$ ,  $g_2(x)$  и целевой

функции  $j(x)$ , определенные на соответствующих множествах  $Q_j$  из (12.6). Изображена также точка  $x^1$  с индексом индекс  $n(x^1)=1$ , точка  $x^2$  с индексом  $n(x^2)=2$ , и точка  $x^3$  с индексом  $n(x^3)=3$ .

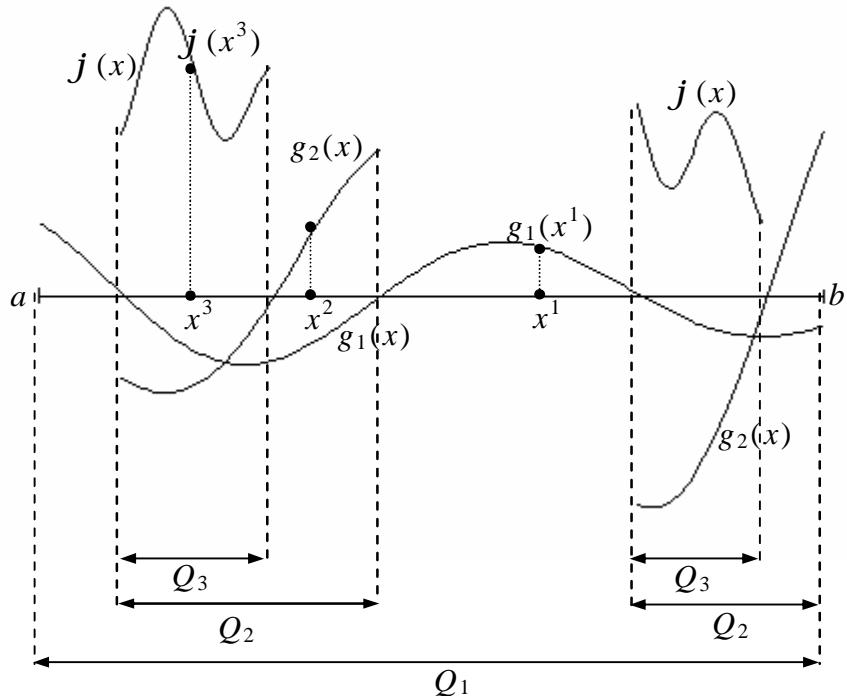


Рис. 12.3. Задача с частично вычислимыми функциями

Данная классификация порождает функцию

$$f(x)=g_n(x), \quad n=n(x), \quad (12.9)$$

определенную и вычислимую всюду в  $[a, b]$ . Ее значение в точке  $x$  есть либо значение левой части ограничения, нарушенного в этой точке (случай, когда  $n \leq m$ ), либо значение минимизируемой функции (случай, когда  $n=m+1$ ). Поэтому определение значения  $f(x)$ ,  $x \in [a, b]$ , сводится к последовательному вычислению величин  $g_j(x)$ ,  $1 \leq j \leq n=n(x)$ , т.е. последующее значение  $g_{j+1}(x)$  вычисляется лишь в том случае, когда  $g_j(x) \leq 0$ . Процесс вычислений завершается либо в результате установления неравенства  $g_j(x) > 0$ , либо в результате достижения значения  $n(x)=m+1$ .

```
// Алгоритм определения индекса
double value; // значение функции
int index; // индекс точки
for( int i = 0; i < m+1; i++ ) {
    value=g[i](x);
    if( (i == m+1) || (value>0) ){
        index = i;
        break;
    }
}
```

Описанная процедура, названная *испытанием* в точке  $x$ , автоматически приводит к определению индекса  $n$  этой точки. Пара значений

$$z=f(x)=g_n(x), \quad n=n(x), \quad (12.10)$$

порожденная испытанием в точке  $x \in [a, b]$ , называется *результатом испытания*.

На рис. 12.4 приведен график функции  $f(x)$  из (12.9), который образован дугами функций ограничений  $g_1(x)$ ,  $g_2(x)$  и целевой функции  $j(x)$  для задачи из примера 12.2.

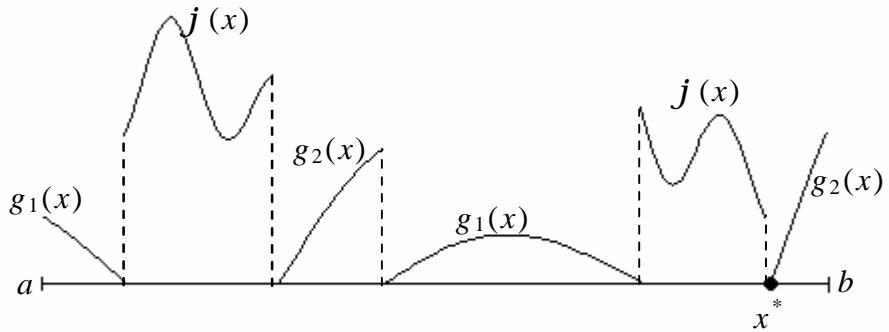


Рис. 12.4. “Индексная” функция

Поскольку в общем случае задача (12.7) может не иметь решения (т.е. допустимая область  $Q_{m+1}$  может оказаться пустой в силу несовместности ограничений), с ней связывается некоторая вспомогательная задача, всегда имеющая решение. Так как условия (12.8) эквивалентны условиям

$$x \in Q_n, x \notin Q_{n+1},$$

то вспомогательная задача, всегда имеющая решение, может быть записана в виде

$$g_M^* = g_M(x_M^*) = \min\{g_M(x): x \in Q_M\}, \quad (12.11)$$

где  $M$  есть максимально возможное значение индекса, т.е.

$$1 \leq M = \max\{n(x): x \in [a, b]\} \leq m+1. \quad (12.12)$$

Поскольку множество  $Q_M$  всегда непусто, задача (12.11) всегда имеет решение. При  $M=m+1$  решение  $x^* = x_{m+1}^*$  является также решением исходной задачи (12.5). При  $M < m+1$  необходимо выполняющееся неравенство  $g_M^* < 0$  может использоваться как индикатор несовместности ограничений.

Основная идея индексного подхода состоит в редукции условной задачи (12.11) к безусловной задаче

$$y(x^*) = \min\{y(x): x \in [a, b]\},$$

где

$$y(x) = \begin{cases} g_n(x)/L_n, & n < M, \\ (g_M - g_M^*)/L_M, & n = M. \end{cases} \quad (12.13)$$

Как результат, дуги функции  $y(x)$  будут липшицевыми с константой  $L=1$  на каждом множестве  $Q_n$ ,  $1 \leq n \leq M$  (на рис. 12.5 приведен график функции  $y(x)$  для задачи из примера 12.2). Эта новая функция будет иметь разрывы первого рода на граничных точках множеств  $Q_n$  из (12.6). Но, тем не менее, можно оценить точку глобального оптимума  $x_M^*$ , используя результаты (12.10)  $k$  испытаний в точках  $x^1, \dots, x^k$  из  $[a, b]$  ([97]).

В самом деле, из условия Липшица следует, что

$$x_M^* \in \{x \in [a, b]: |x - x^i| \geq y(x^i), 1 \leq i \leq k\}. \quad (12.14)$$

Так, например, на рис. 12.5 изображен случай  $k=4$ . Объединение отрезков, выделенных на рисунке жирной линией, не содержит оптимальную точку.

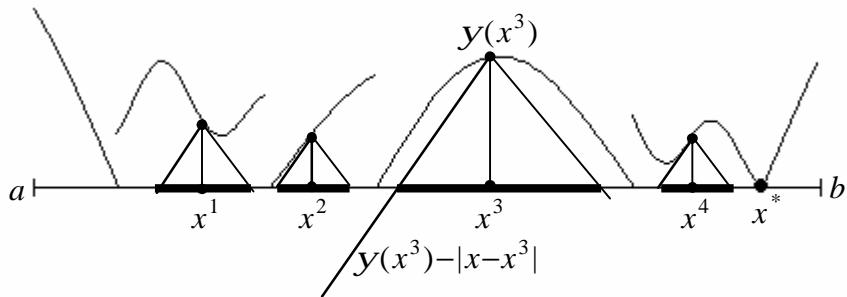


Рис. 12.5. Оценка оптимальной точки

Заметим, что максимальный индекс  $M$ , значения констант Липшица  $L_n$ ,  $1 \leq n \leq M$ , и величина  $g_M^*$  являются неизвестными. Однако эти проблемы можно преодолеть, используя вместо этих величин их адаптивные оценки, получаемые в процессе решения задачи на основании результатов испытаний.

После того, как введены все требуемые понятия, перейдем к изложению индексного алгоритма.

#### 12.4.2. Схема алгоритма

Первое испытание осуществляется в произвольной внутренней точке  $x^1 \in (a, b)$ . Выбор точки  $x^{k+1}$ ,  $k \geq 1$ , любого последующего испытания определяется следующими правилами.

*Правило 1.* Перенумеровать точки  $x^1, \dots, x^k$  предшествующих испытаний нижними индексами в порядке увеличения значений координаты, т.е.

$$a = x_0 < x_1 < \dots < x_i < \dots < x_k < x_{k+1} = b, \quad (12.15)$$

и сопоставить им значения  $z_i = g_n(x_i)$ ,  $n = n(x_i)$ ,  $1 \leq i \leq k$ , из (12.10), вычисленные в этих точках; точки  $x_0 = a$  и  $x_{k+1} = b$  введены дополнительно (значения  $z_0$  и  $z_{k+1}$  не определены) для удобства последующих обозначений.

*Правило 2.* Провести классификацию номеров  $i$ ,  $1 \leq i \leq k$ , точек из ряда (12.15) по числу ограничений задачи, выполняющихся в этих точках, путем построения множеств

$$I_n = \{i : 1 \leq i \leq k, n = n(x_i)\}, \quad 1 \leq n \leq m+1.$$

содержащих номера всех точек  $x_i$ ,  $1 \leq i \leq k$ , имеющих индексы, равные одному и тому же значению  $n$ . Границные точки  $x_0 = a$  и  $x_{k+1} = b$  интерпретируются как имеющие нулевые индексы, и им сопоставляется дополнительное множество  $I_0 = \{0, k+1\}$ .

Определить максимальное значение индекса

$$V = \max \{n = n(x_i), 1 \leq i \leq k\}. \quad (12.16)$$

*Правило 3.* Вычислить текущие нижние границы

$$m_n = \max \left\{ \frac{|z_i - z_j|}{x_i - x_j}, \quad i, j \in I_n, \quad i > j \right\} \quad (12.17)$$

для неизвестных констант Липшица  $L_n$  функций  $g_n$ ,  $1 \leq n \leq m+1$ . Если множество  $I_n$  содержит менее двух элементов или если  $m_n$  из (12.17) оказывается равным нулю, то

принять  $m_n=1$ . Из (12.17) следует, что оценки  $m_n$  являются неубывающими, начиная с момента, когда (12.17) порождает первое положительное значение  $m_n$ .

*Правило 4.* Для всех непустых множеств  $I_n$ ,  $1 \leq n \leq m+1$ , вычислить оценки

$$z_n^* = \begin{cases} 0, & n < V, \\ \min\{g_n(x_i) : i \in I_n\}, & n = V. \end{cases} \quad (12.18)$$

*Правило 5.* Для каждого интервала  $(x_{i-1}, x_i)$ ,  $1 \leq i \leq k+1$ , вычислить характеристику  $R(i)$ , где

$$\begin{aligned} R(i) &= \Delta_i + \frac{(z_i - z_{i-1})^2}{r_n^2 m_n^2 \Delta_i} - 2 \frac{(z_i + z_{i-1} - 2z_n^*)}{r_n m_n}, \quad n = n(x_{i-1}) = n(x_i), \\ R(i) &= 2\Delta_i - 4 \frac{(z_i - z_n^*)}{r_n m_n}, \quad n = n(x_i) > n(x_{i-1}), \\ R(i) &= 2\Delta_i - 4 \frac{(z_{i-1} - z_n^*)}{r_n m_n}, \quad n = n(x_{i-1}) > n(x_i), \\ \Delta_i &= x_i - x_{i-1}. \end{aligned} \quad (12.19)$$

Величины  $r_n > 1$ ,  $1 \leq n \leq m+1$ , являются параметрами алгоритма. Подходящий выбор значений  $r_n$  позволяет использовать произведение  $r_n m_n$  как оценку константы Липшица  $L_n$ ,  $1 \leq n \leq m+1$ .

*Правило 7.* Определить интервал  $(x_{t-1}, x_t)$ , которому соответствует максимальная характеристика

$$R(t) = \max\{R(i) : 1 \leq i \leq k+1\}. \quad (12.20)$$

*Правило 8.* Провести очередное испытание в серединной точке интервала  $(x_{t-1}, x_t)$ , если индексы его концевых точек не совпадают, т.е.

$$x^{k+1} = \frac{x_t + x_{t-1}}{2}, \quad n(x_{t-1}) \neq n(x_t).$$

В противном случае провести испытание в точке

$$x^{k+1} = \frac{x_t + x_{t-1}}{2} - \frac{z_t - z_{t-1}}{2r_n m_n}, \quad n = n(x_{t-1}) = n(x_t). \quad (12.21)$$

Описанные правила можно дополнить условием остановки, прекращающим испытания, если

$$x_t - x_{t-1} \leq e, \quad (12.22)$$

где  $t$  из (12.20) и  $e > 0$  есть заданная точность.

**Условия сходимости алгоритма.** Если рассмотренный индексный алгоритм применяется для решения задачи (12.5), и при этом выполняются условия:

1. каждая функция  $g_j$ ,  $1 \leq j \leq m+1$ , удовлетворяет на отрезке  $[a, b]$  условию Липшица с константой  $L_j$ , т.е.

$$|g_j(x_1) - g_j(x_2)| \leq L_j |x_1 - x_2|, \quad 1 \leq j \leq m+1, \quad x_1, x_2 \in [a, b];$$

2. для величин  $m_n$  из (12.17), начиная с некоторого шага, справедливы неравенства

$$r_n m_n > 2L_n, \quad 1 \leq n \leq m+1.$$

то множество предельных точек последовательности  $\{x^k\}$ , порождаемой индексным алгоритмом, совпадет с множеством решений задачи (12.5) при  $e=0$  в условии остановки (12.22), причем индекс каждой предельной точки равен  $M$ .

### 12.4.3. Программная реализация

Представим возможный вариант программной реализации индексного метода для решения одномерных задач условной оптимизации. В отличие от программ предыдущих разделов разработка выполнена на C++.

Заголовочный файл Method.h содержит объявление класса CMETHOD, реализующего индексный метод, а также объявление всех требуемых для работы структур и типов данных.

```
#include <vector>
#include <set>
#define _USE_MATH_DEFINES
#include <math.h>

#define MaxFuncs 10
typedef double (*pFunc)(double);

// Точка испытания
struct CTrial {
    double Value;
    int index;
    double x;
};

inline bool operator<(const CTrial& t1,const CTrial& t2){return (t1.x<t2.x);}

typedef std::set<CTrial>::iterator pTrial;

// Индексный метод
class CMETHOD {
public:
    // Левая граница интервала поиска
    double a;
    // Правая граница интервала поиска
    double b;
    // Точность поиска
    double eps;
    // Параметр надежности
    double r[MaxFuncs];
    // Число функций задачи
    int NumFuncs;
    // Конструктор по умолчанию
    CMETHOD(){}
    // Деструктор по умолчанию
    ~CMETHOD(){}
    // Запуск метода
    void Run();
    // Функции задачи
    pFunc Funcs[MaxFuncs];
    // Лучшее испытание
    CTrial BestTrial;
private:
    // Параметр Z алгоритма
    double Z[MaxFuncs];
    // Относительные первые разности
    double M[MaxFuncs];
}
```

```

// Указатели на точки испытаний с фиксированным индексом
std::vector<pTrial> I[MaxFuncs];
// Точки испытаний
std::set<CTrial> Trials;
// Максимальный индекс
int MaxIndex;
// Инициализация процесса поиска
void Init();
// Определение множества I
void CalculateI(void);
// Вычисление значений относительных разностей
void CalculateM();
// Вычисление значений Z
void CalculateZ();
// Поиск интервала с максимальной характеристикой
pTrial FindMaxR(void);
// Проведение очередного испытания
CTrial MakeTrial(double);
// Вставка результатов очередного испытания
bool InsertTrial(CTrial);
};

```

Файл *Method.cpp* содержит определение функций класса *CMethod*, реализующего индексный метод.

**1. Функция *Run*** – основная функция класса. Реализует схему индексного алгоритма, вызывает необходимые подпрограммы.

```

// Главная функция индексного метода
void CMethod::Run(){
    // Инициализация процесса поиска
    Init();
    // Условие остановки не выполнено
    bool stop=false;
    CTrial trial;
    while(!stop){
        // Определение множества I
        CalculateI();
        // Вычисление значений относительных разностей
        CalculateM();
        // Вычисление значений Z
        CalculateZ();
        // Поиск интервала с максимальной характеристикой
        pTrial t=FindMaxR();
        // Проведение испытания в интервале № t
        pTrial t1=t;
        t1--;
        if(t->index!=t1->index){
            trial=MakeTrial(0.5*(t->x+t1->x));
        }else{
            trial=MakeTrial(0.5*(t->x+t1->x)-(t->Value-t1->Value)/
                (M[t->index]*2*r[t->index]));
        }
        // Вставка результатов очередного испытания
        stop=InsertTrial(trial);
    }
}

```

**2. Функция *Init*** проводит начальную инициализацию текущих параметров алгоритма (максимального индекса и максимальных значений относительных первых разностей), а также выполняет вставку в массив испытаний граничных точек отрезка  $[a,b]$  и проводит первое испытание в серединной точке отрезка.

```

// Инициализация процесса поиска
void CMethod::Init(){

```

```

// Сначала максимальный индекс не определен
MaxIndex=-1;
// Первоначальное формирование информационных множеств
for(int v=0;v<NumFuncs;v++)M[v]=1;
// Заносим в массив испытаний граничные точки
CTrial trial;
trial.x=a;
trial.index=-1;
InsertTrial(trial);
trial.x=b;
trial.index=-1;
InsertTrial(trial);
// Проводим испытание во внутренней точке, например, в серединной
trial=MakeTrial((a+b)/2);
InsertTrial(trial);
}

```

**3. Функция *CalculateI*** формирует множества  $I_n$ ,  $1 \leq n \leq m+1$ , испытаний с фиксированным индексом  $n$  из (12.16).

```

// Определение множества I
void CMethod::CalculateI(void){
    // Предварительная очистка множества I
    for(int v=0;v<NumFuncs;v++){
        I[v].clear();
    }
    // Заполнение множества I
    for(int v=0;v<NumFuncs;v++){
        pTrial i;
        for(i=Trials.begin();i!=Trials.end();i++){
            if(i->index==v){
                I[v].push_back(i);
            }
        }
    }
}

```

**4. Функция *CalculateM*** выполняет вычисление максимального значения относительных первых разностей  $m_n$  из (12.17).

```

// Вычисление значений относительных разностей
void CMethod::CalculateM(){
    for(int v=0;v<NumFuncs;v++){
        if(I[v].size()<2){
            M[v]=1;
            continue;
        }
        double MaxM=-HUGE_VAL;
        double tM;
        for(unsigned i=1;i<I[v].size();i++){
            tM=fabs(I[v][i]->Value-I[v][i-1]->Value)/(I[v][i]->x-I[v][i-1]->x);
            if(MaxM<tM)MaxM=tM;
        }
        if(MaxM>0)M[v]=MaxM;
        else M[v]=1;
    }
}

```

**5. Функция *CalculateZ*** выполняет вычисление оценок  $z_n$  из (12.18).

```

// Вычисление значений Z
void CMethod::CalculateZ(){
    for(int v=0;v<NumFuncs;v++){
        if(v<MaxIndex){
            Z[v]=0;
        }
    }
}

```

```

        continue;
    }
    Z[v]=BestTrial.Value;
    break;
}
}

```

**6. Функция *FindMaxR*** выполняет поиск интервала с максимальной характеристикой и возвращает указатель на этот интервал.

```

// Поиск интервала с максимальной характеристикой
pTrial CMethod::FindMaxR(void){
    pTrial t;
    pTrial i1=Trials.begin();
    pTrial i=Trials.begin();
    i++;
    double MaxR=-HUGE_VAL;
    double R;
    // Вычисление характеристик
    for(i;i!=Trials.end();i++,i1++){
        double deltax=i->x-i1->x;
        if(i->index == i1->index){
            int v=i->index;
            R=deltax+(i->Value-i1->Value)*(i->Value-i1->Value)/
                (deltax*M[v]*M[v]*r[v]*r[v])-2*(i->Value+i1->Value-2*Z[v])/
                (r[v]*M[v]);
        }
        if(i->index>i1->index){
            int v=i->index;
            R=2*deltax-4*(i->Value-Z[v])/(r[v]*M[v]);
        }
        if(i->index<i1->index){
            int v=i1->index;
            R=2*deltax-4*(i1->Value-Z[v])/(r[v]*M[v]);
        }
        if(R>MaxR){
            MaxR=R;
            t=i;
        }
    }
    return t;
}

```

**7. Функция *MakeTrial*** выполняет проведение очередного испытания в точке  $x$  и возвращает результаты испытания в виде структуры *CTrial*.

```

// Проведение очередного испытания
CTrial CMethod::MakeTrial(double x){
    CTrial Trial;
    Trial.x=x;
    // Проверка ограничений
    for(int i=0;i<NumFuncs;i++){
        Trial.Value=Funcs[i](x);
        if(i==NumFuncs-1||Trial.Value>0){
            Trial.index=i;
            break;
        }
    }
    return Trial;
}

```

**8. Функция *InsertTrial*** выполняет вставку результатов очередного испытания в упорядоченное множество испытаний. В случае выполнения критерия остановки функция возвращает *true*, иначе – *false*.

```

// Вставка результатов очередного испытания
bool CMethod::InsertTrial(CTrial Trial){
    // Вставка результатов
    std::pair<pTrial, bool> ins;
    ins=Trials.insert(Trial);
    // Точки итераций совпали - стоп
    if(!ins.second) return true;
    if(Trials.size()>2){
        pTrial j=ins.first, j1=ins.first;
        j++;
        j1--;
        // Выполнено условие остановки
        if((j->x-j1->x)<eps) return true;
    }
    // Оценка значений
    if(Trial.index>MaxIndex || Trial.index==MaxIndex && Trial.Value<BestTrial.Value){
        BestTrial=Trial;
        MaxIndex=Trial.index;
    }
    return false;
}

```

#### 12.4.4. Результаты численных экспериментов

В качестве иллюстрации рассмотрим задачу из примера 12.2:  $x \in [0.6, 2.2]$ ,

$$j(x) = \cos(18x-3)\sin(10x-7)+1.5,$$

$$g_1(x) = \exp(-x/2)\sin(6x-1.5),$$

$$g_2(x) = |x|\sin(2\pi x-0.5).$$

В предположении частичной вычислимости дуги этих функций, соответствующие областям  $Q_j$ ,  $1 \leq j \leq 2$ , из (12.6), представлены на рис. 12.6.

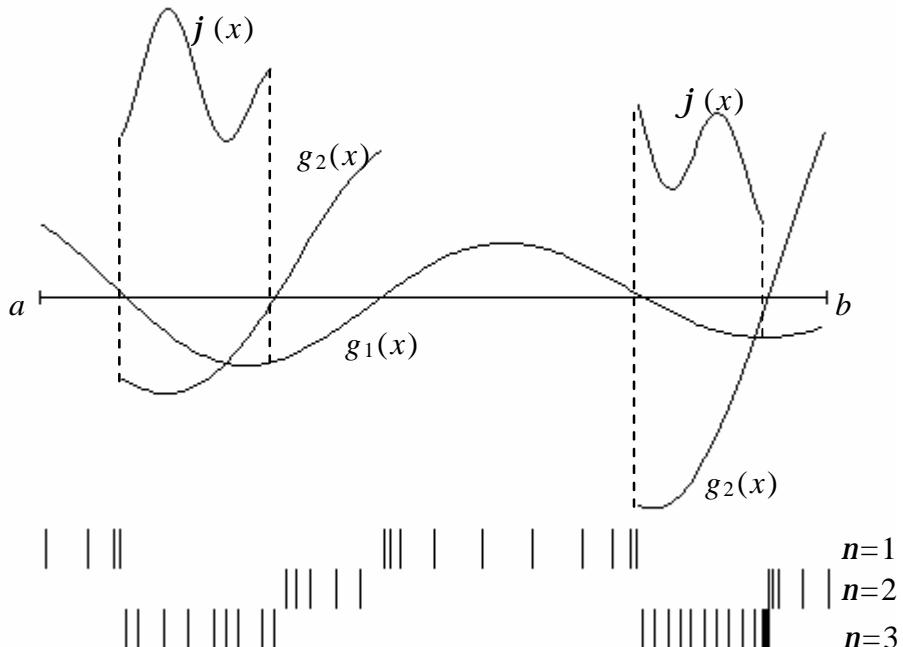


Рис. 12.6. Результаты решения задачи

Описанный индексный алгоритм был использован для решения этого примера при  $r_n=2$ ,  $1 \leq n \leq 3$ , и  $e=10^{-5}$  (исходный код программы см. ниже). Координаты точек испытаний, осуществленных алгоритмом в процессе решения задачи, отмечены на

рис. 12.6 тремя рядами вертикальных штрихов. Штрихи верхнего ряда соответствуют точкам с единичным индексом, второго – точкам, индексы которых равны 2; точки, отмеченные штрихами нижнего ряда, являются допустимыми. Координаты испытаний, выполненных в близких точках, отмечены темным прямоугольником.

Следует отметить, что весьма часто в прикладных задачах оценка значений функций ограничений и целевой функции требует заметных вычислительных ресурсов. Результаты проведенного эксперимента подтверждают экономичность используемой индексной схемы учета ограничений, т.к. при этом значения функций  $g_1, g_2, j=g_3$  вычислялись соответственно  $k_1=63, k_2=49$  и  $k_3=35$  раз. Если бы поиск решения проводился бы на равномерной сетке, то для достижения такой же точности  $e=10^{-5}$  потребовалось бы  $k_1=1.6 \cdot 10^5, k_2=90280$  и  $k_3=56476$  вычислений значений функций  $g_1, g_2, j=g_3$  соответственно.

```
// Решение тестового примера индексным методом
#include <iostream>
#include "method.h"
// Первое ограничение
double f1(double x){
    return exp(-0.5*x)*sin(6*x-1.5);
}
// Второе ограничение
double f2(double x){
    return fabs(x)*sin(2*M_PI*x-0.5);
}
// Целевая функция
double fi(double x){
    return cos(18*x-3)*sin(10*x-7)+1.5;
}

int main(int argc, char* argv[]){
    CMethod im;
    im.a=0.6;
    im.b=2.2;
    im.eps=0.00001;
    im.NumFuncs=3;
    im.Funcs[0]=f1;
    im.Funcs[1]=f2;
    im.Funcs[2]=fi;
    for(int i=0;i<3;i++)im.r[i]=2;
    im.Run();
    std::cout<<"Xmin="<<im.BestTrial.x<<"Ymin="<<im.BestTrial.Value<<std::endl;
    return 0;
}
```

## 12.5. Редукция размерности задачи

### 12.5.1. Использование отображений Пеано

Рассмотрим многомерную задачу глобальной оптимизации вида

$$\begin{aligned} j(w^*) = \min\{j(w) : w \in S, g_j(w) \leq 0, 1 \leq j \leq m\}, \\ S = \{s \in R^N : a_i \leq s_i \leq b_i, 1 \leq i \leq N\}. \end{aligned}$$

Нетрудно видеть, что с помощью преобразования

$$y_i = (w_i - (a_i + b_i)/2)/r, \quad r = \max\{b_i - a_i : 1 \leq i \leq N\},$$

и введения дополнительного ограничения

$$g_0(y) = \max\{|y_i| - (b_i - a_i)/2r : 1 \leq i \leq N\} \leq 0,$$

можно представить исходную задачу условной глобальной оптимизации, определенную на гиперинтервале  $S$ , как задачу на единичном  $N$ -мерном гиперкубе  $D$ :

$$\begin{aligned} j(y^*) &= \min\{j(y) : y \in D, g_j(y) \leq 0, 1 \leq j \leq m\}, \\ D &= \{y \in R^N : -2^{-1} \leq y_i \leq 2^{-1}, 1 \leq i \leq N\} = \{y(x) : 0 \leq x \leq 1\}. \end{aligned} \quad (12.23)$$

Использование развертки Пеано  $y(x)$ , однозначно отображающей единичный отрезок вещественной оси на единичный гиперкуб, позволяет свести многомерную задачу условной минимизации в области  $D$  к одномерной задаче условной минимизации на отрезке  $[0, 1]$

$$j(y(x^*)) = \min\{j(y(x)) : x \in [0, 1], g_j(y(x)) \leq 0, 1 \leq j \leq m\}. \quad (12.24)$$

Рассматриваемая схема редукции размерности сопоставляет многомерной задаче с липшицевой минимизируемой функцией и липшицевыми ограничениями левыми частями ограничений одномерную задачу, в которой соответствующие функции удовлетворяют равномерному условию Гельдера (см. [28,97]), т.е.

$$|g_j(x') - g_j(x'')| \leq K_j |x' - x''|^{1/N}, \quad x', x'' \in [0, 1], \quad 1 \leq j \leq m+1,$$

где  $N$  есть размерность исходной многомерной задачи, а коэффициенты  $K_j$  связаны с константами Липшица  $L_j$  исходной задачи соотношениями  $K_j \leq 4L_j\sqrt{N}$ . Следовательно, все длины интервалов, участвующие в правилах алгоритма поиска, должны быть заменены на длины в новой метрике, в которой расстояние определяется выражением

$$\Delta_i = |x_i - x_{i-1}|^{1/N}.$$

Для этого заменим в выражениях (12.17), (12.19) и (12.22) разности  $x_i - x_{i-1}$  величинами  $|x_i - x_{i-1}|^{1/N}$ , а вместо (12.21) введем выражение

$$x^{k+1} = (x_t + x_{t-1})/2 - \text{sign}(z_t - z_{t-1}) \frac{1}{2r_n} \left[ \frac{|z_t - z_{t-1}|}{m_n} \right]^N, \quad n = n(x_{t-1}) = n(x_t).$$

Как результат получаем алгоритм решения задачи (12.24), описание которого в целом совпадает со схемой одномерного индексного алгоритма (см. [94,97]).

Вопросы численного построения отображений типа кривой Пеано и соответствующая теория подробно рассмотрены в работах [28,97]. Здесь же отметим, что численно построенная кривая является приближением к теоретической кривой Пеано с точностью, не хуже  $2^{-m}$  по каждой координате (параметр  $m$  называется *плотностью развертки*).

### 12.5.2. Программная реализация

Представим для рассмотрения функции, реализующие отображение типа кривой Пеано.

**1. Функция *mapd*** реализует прямое отображение: для заданной точки  $x \in [0, 1]$ , плотности развертки  $m$  и размерности пространства  $n$  вычисляется образ  $y(x) \in D$ .

```
// Реализация разверток
int n1, nexp, l, iq, iu[10], iv[10];
// Прямое отображение
void mapd( double x, int m, float* y, int n ) {
    double d, mne, dd, dr, tmp;
    float p, r;
```

```

int iw[11];
int it,is,i,j,k;

p=0.0;
n1=n-1;
for ( nexp=1,i=0; i<n; nexp*=2,i++ );
d=x; r=0.5; it=0; dr=nexp;
for ( mne=1,i=0; i<m; mne*=dr,i++ );
for ( i=0; i<n; i++ ) { iw[i]=1; y[i]=0.0; }
for ( j=0; j<m; j++ ) {
    iq=0;
    if ( x == 1.0 ) {
        is=nexp-1; d=0.0;
    } else {
        d=d*nexp;
        is=d;
        d=d-is;
    }
    i=is;
    node(i);
    i=iu[0];
    iu[0]=iu[it];
    iu[it]=i;
    i=iv[0];
    iv[0]=iv[it];
    iv[it]=i;
    if ( l == 0 ) l=it;
    else if ( l == it ) l=0;
    if ( (iq>0) | ((iq==0)&&(is==0)) ) k=l;
    else if ( iq<0 ) k = ( it==n1 ) ? 0 : n1;
    r=r*0.5;
    it=l;
    for ( i=0; i<n; i++ ) {
        iu[i]=iu[i]*iw[i];
        iw[i]=-iv[i]*iw[i];
        p=r*iu[i];
        p=p+y[i];
        y[i]=p;
    }
}
}
// Вспомогательная функция
void node ( int is ) {
    int n,i,j,k1,k2,iff;

    n=n1+1;
    if ( is == 0 ) {
        l=n1;
        for ( i=0; i<n; i++ ) { iu[i]=-1; iv[i]=-1; }
    } else if ( is == (nexp-1) ) {
        l=n1;
        iu[0]=1;
        iv[0]=1;
        for ( i=1; i<n; i++ ) { iu[i]=-1; iv[i]=-1; }
        iv[n1]=1;
    } else {
        iff=nexp;
        k1=-1;
        for ( i=0; i<n; i++ ) {
            iff=iff/2;
            if ( is >= iff ) {
                if ( (is==iff)&&(is != 1) ) { l=i; iq=-1; }
                is=is-iff;
                k2=1;
            }
        }
    }
}

```

```

    }
    else {
        k2=-1;
        if ( (is==(iff-1))&&(is!= 0) ) { l=i; iq=1; }
    }
    j=-k1*k2;
    iv[i]=j;
    iu[i]=j;
    k1=k2;
}
iv[l]=iv[l]*iq;
iv[nl]=-iv[nl];
}
}

```

**2. Функция *xyd*** реализует обратное отображение: для заданной точки  $y \in D$ , плотности развертки  $m$  и размерности пространства  $n$  вычисляется ее прообраз  $y^{-1}(x) \in [0,1]$ .

```

// Обратное отображение
void xyd ( double *xx, int m, float y[], int n) {
    double x,r1;
    float r;
    int iw[10];
    int i,j,it,is;

    n1=n-1;
    for ( nexp=1,i=0; i<n; i++ ) { nexp*=2; iw[i]=1; }
    r=0.5; r1=1.0; x=0.0; it=0;
    for ( j=0; j<m; j++ ) {
        r*=0.5;
        for ( i=0; i<n; i++ ) {
            iu[i] = ( y[i]<0 ) ? -1 : 1;
            y[i]-=r*iu[i];
            iu[i]*=iw[i];
        }
        i=iu[0];
        iu[0]=iu[it];
        iu[it]=i;
        numbr ( &is );
        i=iv[0];
        iv[0]=iv[it];
        iv[it]=i;
        for ( i=0; i<n; i++ ) iw[i]=-iw[i]*iv[i];
        if ( l == 0 ) l=it;
        else if ( l == it ) l=0;
        it=l;
        r1=r1/nexp;
        x+=r1*is;
    }
    *xx=x;
}
// Вспомогательная функция
void numbr ( int *iss) {
    int i,n,is,iff,k1,k2,l1;

    n=n1+1; iff=nexp;
    is=0; k1=-1;
    for ( i=0; i<n; i++ ) {
        iff=iff/2;
        k2=-k1*iu[i];
        iv[i]=iu[i];
        k1=k2;
        if ( k2<0 ) l1=i;
    }
}

```

```

    else { is+=iff; l=i; }
}
if ( is == 0 ) l=n1;
else {
    iv[n1]=-iv[n1];
    if ( is == (nexp-1) ) l=n1;
    else if ( l1 == n1 ) iv[l]=-iv[l];
    else l=l1;
}
*iss=is;
}

```

### 12.5.3. Результаты численных экспериментов

В качестве иллюстрации рассмотрим задачу из примера 12.1: минимизировать функцию

$$j(y_1, y_2) = -1.5y_1^2 \exp\{1 - y_1^2 - 20.25(y_1 - y_2)^2\} - \\ - [0.5(y_1 - 1)(y_2 - 1)]^4 \exp\{2 - [0.5(y_1 - 1)]^4 - (y_2 - 1)^4\}$$

в области поиска  $0 \leq y_1 \leq 4$ ,  $-1 \leq y_2 \leq 3$ , при ограничениях

$$g_1(y_1, y_2) = 0.01[(y_1 - 2.2)^2 + (y_2 - 1.2)^2 - 2.25] \leq 0$$

$$g_2(y_1, y_2) = 100[1 - (y_1 - 2)^2 / 1.44 - (0.5y_2)^2] \leq 0$$

$$g_3(y_1, y_2) = 10[y_2 - 1.5 - 1.5 \sin(6.283(y_1 - 1.75))] \leq 0$$

Допустимые по первому ограничению точки образуют круг с границей  $g_1(y_1, y_2)=0$ . Допустимые по второму ограничению точки находятся во внешности эллипса  $g_2(y_1, y_2)=0$ . Точки, допустимые по третьему ограничению, находятся ниже синусоиды  $g_3(y_1, y_2)=0$ . Глобальный минимум  $j(y_1^*, y_2^*) = -1.489$  достигается в точке  $(y_1^*, y_2^*) = (0.942, 0.944)$ .

На рисунке 12.7 изображена область поиска – квадрат, где обозначены точки 1098 испытаний, полученные при применении индексного метода со следующими параметрами: параметр надежности  $r_1 = \dots = r_4 = 2$ , точность решения  $e = 10^{-3}$ , плотность построения развертки  $m = 12$ .

Результаты проведенного эксперимента подтверждают экономичность предлагаемого подхода, т.к. при этом значения функций  $g_1, g_2, g_3, j=g_4$  вычислялись соответственно  $k_1=1098$ ,  $k_2=623$ ,  $k_3=392$  и  $k_4=152$  раз. Если бы поиск решения проводился бы на равномерной сетке, то для достижения такой же точности  $e=10^{-3}$  потребовалось бы  $k_1=1.6 \cdot 10^7$ ,  $k_2=7 \cdot 10^6$   $k_3=3 \cdot 10^6$  и  $k_4=1.4 \cdot 10^6$  вычислений значений функций  $g_1, g_2, g_3, j=g_4$  соответственно.

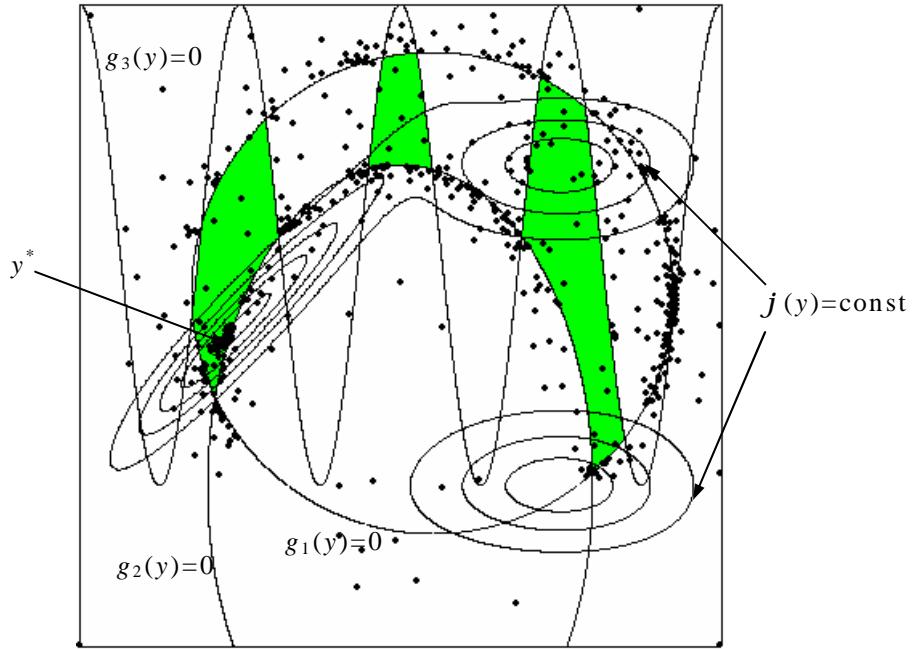


Рис. 12.7. Результаты решения задачи

#### 12.5.4. Способ построения развертки

Для желающих подробно познакомиться со способом построения кривых Пеано опишем схему построения развертки (см. [28,97]), отображающей единичный отрезок вещественной оси  $[0,1]$  на гиперкуб  $D$ ,

$$D = \{y \in R^N : -2^{-1} \leq y_i \leq 2^{-1}, 1 \leq i \leq N\}.$$

1. Гиперкуб  $D$ , длина ребра которого равна 1, разделяется координатными плоскостями на  $2^N$  гиперкубов первого разбиения (с длиной ребра, равной  $1/2$ ), которые нумеруются числами  $z_1$  от 0 до  $2^N - 1$ , причем гиперкуб первого разбиения с номером  $z_1$  условимся обозначать через  $D(z_1)$ .

Далее, каждый гиперкуб первого разбиения, в свою очередь, также разбивается на  $2^N$  гиперкубов второго разбиения (с длиной ребра, равной  $1/4$ ) гиперплоскостями, параллельными координатным осям и проходящими через серединные точки ребер гиперкуба, ортогональных к этим гиперплоскостям. При этом гиперкубы второго разбиения, входящие в гиперкуб  $D(z_1)$ , нумеруются числами  $z_2$  от 0 до  $2^N - 1$ , причем гиперкуб второго разбиения с номером  $z_2$ , входящий в  $D(z_1)$ , обозначается через  $D(z_1, z_2)$ . Случай  $N=2$  изображен на рис. 12.8 для  $m=1$ ,  $m=2$ .

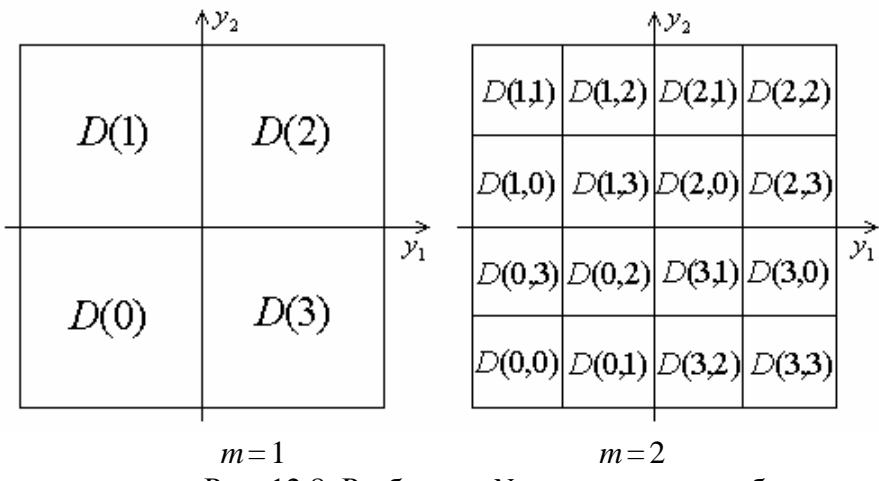


Рис. 12.8. Разбиение  $N$ -мерного гиперкуба

Продолжая указанный процесс, можно построить гиперкубы любого  $m$ -го разбиения с длиной ребра, равной  $(1/2)^m$ , которые обозначаются как  $D(z_1, \dots, z_m)$ , причем  $D(z_1) \supset D(z_1, z_2) \supset \dots \supset D(z_1, \dots, z_m)$  и  $0 \leq z_j \leq 2^N - 1$ ,  $1 \leq j \leq m$ .

**2.** Теперь осуществим деление отрезка  $[0, 1]$  на  $2^N$  равных частей, каждую из которых, в свою очередь, также разделим на  $2^N$  равных частей и так далее, причем элементы каждого разбиения нумеруются слева направо числами  $z_j$  от 0 до  $2^N - 1$ , где  $j$  – номер разбиения. При этом интервалы  $m$ -го разбиения обозначим как  $d(z_1, \dots, z_m)$ , где, например,  $d(z_1, z_2)$  обозначает интервал второго разбиения с номером  $z_2$ , являющийся частью интервала  $d(z_1)$  первого разбиения с номером  $z_1$ .

Можно отметить, что  $d(z_1) \supset d(z_1, z_2) \supset \dots \supset d(z_1, \dots, z_m)$ , и длина интервала  $d(z_1, \dots, z_m)$  равна  $(1/2)^{mN}$ . Предполагается, что интервал  $d(z_1, \dots, z_m)$  содержит свой левый конец. Он содержит правый конец тогда и только тогда, когда  $z_1 = z_2 = \dots = z_m = 2^N - 1$ . Случай  $N=2$  изображен на рис. 12.9 для  $m=1$  и  $m=2$ .

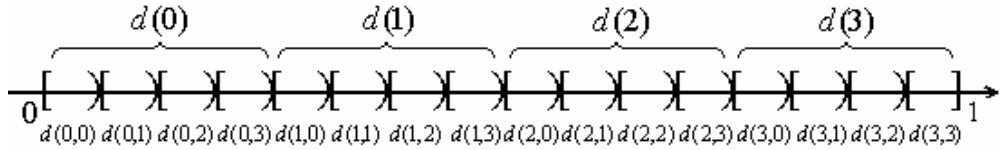


Рис. 12.9. Разбиение одномерного отрезка

**3.** Примем, что точка  $y(x) \in D$ , соответствующая точке  $x \in [0, 1]$ , при любом  $m \geq 1$  содержится в гиперкубе  $D(z_1, \dots, z_m)$ , если  $x$  принадлежит интервалу  $d(z_1, \dots, z_m)$ , т. е.

$$x \in d(z_1, \dots, z_m) \rightarrow y(x) \in D(z_1, \dots, z_m).$$

Построенное соответствие  $y(x)$  является однозначным.

**4.** Чтобы построенное отображение было еще и непрерывным, наложим требования на порядок нумерации гиперкубов каждого разбиения. Так как  $2^{mN}$  центров  $y(z_1, \dots, z_m)$  гиперкубов  $m$ -го разбиения  $D(z_1, \dots, z_m)$  образуют равномерную ортогональную сетку в области  $D$  (причем шаг этой сетки по любой координате равен  $2^{-m}$ ), то можно установить следующую нумерацию узлов этой сетки. Пронумеруем слева направо по  $i$  все интервалы, составляющие  $m$ -е разбиение отрезка  $[0, 1]$ , т. е.

$$d(z_1, \dots, z_m) = [x_i, x_{i+1}), \quad 0 \leq i < 2^{mN} - 1,$$

где через  $x_i$  обозначен левый конец интервала, имеющего номер  $i$ .

Будем считать, что центр гиперкуба  $D(z_1, \dots, z_m)$  имеет тот же номер  $i$ , что и соответствующий этому гиперкубу интервал  $d(z_1, \dots, z_m)$ , т. е.

$$y_i = y(z_1, \dots, z_m), \quad 0 \leq i < 2^{mN} - 1.$$

При этом центры  $y_i$  и  $y_{i+1}$  соответствуют смежным гиперкубам, имеющим общую грань.

Рассмотрим отображение  $l(x)$  отрезка  $[0, 1]$  в гиперкуб  $D$ , определяемое выражениями

$$l(x) = y_i + (y_{i+1} - y_i) \left( \frac{w(x) - x_i}{x_{i+1} - x_i} \right), \quad x_i \leq w(x) \leq x_{i+1},$$

$$w(x) = x(1 - 2^{-mN}), \quad 0 \leq x \leq 1.$$

Образ любого подынтервала отрезка  $[0, 1]$  вида

$$[x_i(1-2^{-mN})^{-1}, x_{i+1}(1-2^{-mN})^{-1}], \quad 0 \leq i < 2^{mN}-1,$$

при соответствии  $l(x)$  является линейным отрезком, соединяющим узлы  $y_i$  и  $y_{i+1}$ .

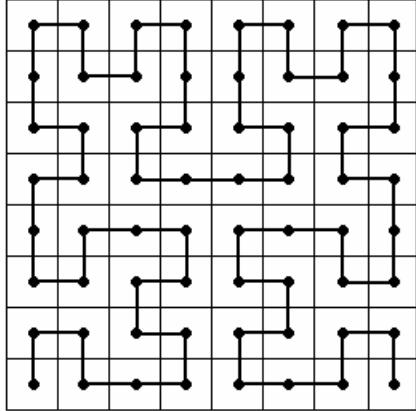


Рис. 12.10. Кусочно-линейная развертка

Таким образом, мы построили кусочно-линейную кривую  $l_m(x)$ ,  $0 \leq x \leq 1$ , соединяющую узлы  $y_i$ ,  $0 \leq i < 2^{mN}-1$ , в порядке их нумерации. Эта кривая является приближением к кривой Пеано с точностью, не хуже  $2^{-m}$  по каждой координате. Для иллюстрации на рис. 12.10 изображен образ отрезка  $[0,1]$  при соответствии  $l(x)$  для случая  $N=2$ ,  $m=3$  (узлы сетки отмечены темными кружками).

## 12.6. Использование множественных отображений

### 12.6.1. Основная схема

Редукция многомерных задач к одномерным с помощью разверток имеет такие важные свойства, как непрерывность и сохранение равномерной ограниченности разностей функций при ограниченности вариации аргумента. Однако при этом происходит частичная потеря части информации о близости точек в многомерном пространстве, ибо точка  $x \in [0,1]$  имеет лишь левых и правых соседей, а соответствующая ей точка  $y(x) \in R^N$  имеет соседей по  $2^N$  направлениям. Как результат, при использовании отображений типа кривой Пеано близким в  $N$ -мерном пространстве образам  $y'$ ,  $y''$  могут соответствовать достаточно далекие прообразы  $x'$ ,  $x''$  на отрезке  $[0,1]$ .

Возможным способом преодоления этого недостатка является использование множественных отображений

$$Y_L(x) = \{y^0(x), y^1(x), \dots, y^L(x)\} \quad (12.25)$$

вместо применения единственной кривой Пеано  $y(x)$  (см. [94,97]).

Каждая кривая Пеано  $y^i(x)$  из  $Y_L(x)$  может быть получена в результате некоторого сдвига вдоль главной диагонали гиперинтервала  $D$ .

Таким образом сконструированное множество кривых Пеано позволяет получить для любых близких образов  $y'$ ,  $y''$ , отличающихся только по одной координате, близкие прообразы  $x'$ ,  $x''$  для некоторого отображения  $y^i(x)$ .

Использование множества отображений приводит к формированию соответствующего множества одномерных многоэкстремальных задач

$$\min\{j(y^l(x)): x \in [0,1], g_j(y^l(x)) \leq 0, 1 \leq j \leq m\}, \quad 0 \leq l \leq L.$$

Каждая задача из данного набора может решаться независимо, при этом любое вычисление значение  $z=g_n(y')$ ,  $n=n(y')$ ,  $y'=y^i(x')$  функции  $g_n(y)$  в  $i$ -й задаче может интерпретироваться как вычисление значения  $z=g_n(y')$ ,  $n=n(y')$ ,  $y'=y^s(x'')$  для любой другой  $s$ -й задачи без повторных трудоемких вычислений функции  $g_n(y)$ . Подобное информационное единство дает возможность решения всего набора задач параллельно.

### 12.6.2. Программная реализация

Представим для рассмотрения функции, реализующие множественные отображения.

**1. Функция *GetImage*** реализует прямое отображение: для заданной точки  $x \in [0, 1]$ , плотности развертки  $m$ , размерности пространства  $n$  и номера развертки  $l$  вычисляется ее образ  $y^l(x)$ .

```
// Вычисление образа точки из отрезка [0,1]
void GetImage (double x, int n, int m, int l, float y[]){
    double del;
    int i;
    if (l == 0) del = 0.0;
    else for (i = 1, del = 1; i < l + 1; del /= 2, i++);
    mapd(x, m+1, y, n, 1);
    for (i = 0; i < n; i++) y[i] = 2 * y[i] + 0.5 - del;
}
```

**2. Функция *GetPreimages*** реализует обратное отображение: для заданной точки  $u$  из многомерного пространства, плотности развертки  $m$ , размерности пространства  $n$  и числа разверток  $L$  вычисляются все ее прообразы  $x^0, x^1, \dots, x^L$ .

```
// Вычисление всех прообразов точки из N-мерного пространства
void GetPreimages( float* p, int n, int m, int L, double xp[]){
    int i, j;
    double xx;
    double del;
    float* p2=new float[n];
    del = 0.5;
    for (i = 1; i < L + 1; i++){
        for (j = 0; j < n; j++) p2[j] = (p[j] + del - 0.5) * 0.5;
        xyd(&xx, m + 1, p2, n);
        xp[i] = xx;
        del *= 0.5;
    }
    del = 0.0;
    for (j = 0; j < n; j++) p2[j] = (p[j] + del - 0.5) * 0.5;
    xyd(&xx, m + 1, p2, n);
    xp[0] = xx;
}
```

### 12.6.3. Способ построения множественных отображений

Материал данного пункта предназначен для желающих подробно ознакомиться со способом построения множественных отображений. Итак, рассмотрим семейство гиперкубов

$$D_l = \{y \in R^N : -2^{-1} \leq y_i + 2^{-l} \leq 3 \cdot 2^{-1}, 1 \leq i \leq N\}, \quad 0 \leq l \leq L, \quad (12.26)$$

где гиперкуб  $D_{l+1}$  получается путем “сдвига” гиперкуба  $D_l$  вдоль главной диагонали на шаг  $2^{-l}$  по каждой координате. На рис. 12.11 изображены гиперкубы  $D_0, \dots, D_3$  для случая  $L=3$ .

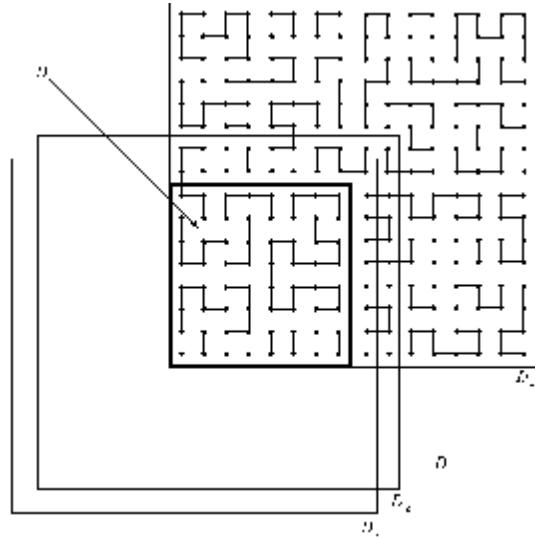


Рис. 12.11. Множественная развертка

Примем, что развертка  $y^0(x)$  типа кривой Пеано отображает отрезок  $[0,1]$  на гиперкуб  $D_0$  из (12.26), т.е.

$$D_0 = \{y^0(x) : x \in [0,1]\}. \quad (12.27)$$

Тогда развертки  $y^l(x) = \{y_1^l(x), \dots, y_N^l(x)\}$ , координаты которых определяются условиями

$$y_i^l(x) = y_i^{l-1}(x) + 2^{-l}, \quad 1 \leq i \leq N, \quad 1 \leq l \leq L, \quad (12.28)$$

отображают отрезок  $[0,1]$  на соответствующие гиперкубы  $D_l, 0 \leq l \leq L$  (на рис. 12.11 ломаной линией изображен образ отрезка  $[0,1]$ , который получается при использовании развертки  $y^0(x), x \in [0,1]$ ). Поскольку гиперкуб  $D$  из (12.23) входит в общую часть семейства гиперкубов семейства (12.26) (граница гиперкуба  $D$  выделена на рис. 12.11), то, введя дополнительную функцию ограничения

$$g_0(y) = \max \{|y_i| - 2^{-1} : 1 \leq i \leq N\}, \quad (12.29)$$

исходный гиперкуб  $D$  можно представить в виде

$$D = \{y^l(x) : x \in [0,1], g_0(y^l(x)) \leq 0\}, \quad 0 \leq l \leq L, \quad (12.30)$$

т.е.  $g_0(y) \leq 0$ , если  $y \in D$ , и  $g_0(y) > 0$  в противном случае. Следовательно, любая точка  $y \in D$  имеет свой прообраз  $x^l \in [0,1]$  при каждом соответствии  $y^l(x), 0 \leq l \leq L$ .

Таким образом, каждая развертка  $y^l(x), 0 \leq l \leq L$ , порождает свою задачу вида (12.4), характеризуемую своей расширенной по сравнению с  $D$  областью поиска  $D_l$  и дополненной ограничением с левой частью из (12.29):

$$\min \{j(y^l(x)) : x \in [0,1], g_j(y^l(x)) \leq 0, 0 \leq j \leq m\}, \quad 0 \leq l \leq L. \quad (12.31)$$

При этом задачам (12.31) соответствуют область  $Q_0 = [0,1]$  и области  $Q'_{j+1}, 0 \leq j \leq m$ , определяемые вторым выражением из (12.6) для соответствующих разверток  $y^l(x)$ . Применение множественного отображения определяет следующую связь окрестностей в многомерных и одномерных областях поиска.

**Утверждение.** Пусть  $y^*$  – произвольная точка из области поиска  $D$ , принадлежащая отрезку с концевыми точками  $y^*, y^* \in D$ , различающимися значениями единственной координаты, и пусть

$$|y'_j - y''_j| \leq 2^{-p}, \quad y'_i = y''_i = y^*_i, \quad 1 \leq i \leq N, \quad i \neq j,$$

где  $p$  – целое число,  $1 \leq p \leq L-1$ , и номер координаты, значения которой для точек  $y^*, y^\ell, y^z$  являются различными, обозначен через  $j$ . Тогда существует хотя бы одно соответствие  $y^l(x)$ ,  $0 \leq l \leq L$ , и прообразы  $x^*, x^\ell, x^z \in [0, 1]$  такие, что

$$\begin{aligned} y^* &= y^l(x^*), \quad y^\ell = y^l(x^\ell), \quad y^z = y^l(x^z), \\ \max \{|x^\ell - x^*|, |x^z - x^*|, |x^\ell - x^z|\} &\leq 2^{-pN}. \end{aligned}$$

Доказательство данного утверждения изложено в работе [97].

**Замечание.** Условия утверждения выделяют специфическую окрестность точки  $y^*$ . Эта окрестность включает в себя лишь такие точки, которые могут быть получены путем смещения  $y^*$  параллельно одной из осей координат на расстояние, не превышающее  $2^{-p}$ . Изменяя значение  $j$ ,  $1 \leq j \leq N$ , в условиях теоремы, можно выделить ближайших “соседей” точки  $y^*$  по  $N$  координатным направлениям. Согласно утверждению, близость точек в  $N$ -мерном пространстве по конкретному направлению будет отражена близостью их прообразов в одной из одномерных подзадач. Сохранение информации о близости точек приводит, во-первых, к более точной оценке констант Липшица, и, во-вторых, к увеличению характеристик интервалов, образы граничных точек которых являются близкими в  $N$ -мерном пространстве.

## 12.7. Параллельный индексный метод

### 12.7.1. Организация параллельных вычислений

Использование множественных отображений позволяет решать исходную задачу (12.1) путем параллельного решения индексным методом  $L+1$  задач вида (12.31) на наборе отрезков  $[0, 1]$ . Каждая одномерная задача (или группа задач при недостаточном количестве процессоров) решается на отдельном процессоре. Результаты испытания в точке  $x^k$ , полученные конкретным процессором для решаемой им задачи, интерпретируются как результаты испытаний во всех остальных задачах (в соответствующих точках  $x^{k0}, x^{k1}, \dots, x^{kL}$ ). При таком подходе испытание в точке  $x^k \in [0, 1]$ , осуществляющееся в  $s$ -й задаче, состоит в последовательности действий:

1. Определить образ  $y^s = y^s(x^k)$  при соответствии  $y^s(x)$ .
2. Вычислить величину  $g_0(y^k)$ . Если  $g_0(y^k) \leq 0$ , то есть  $y^k \in D$ , то проинформировать остальные процессоры о начале проведения испытания в точке  $y^k$  (*блокирование* точки  $y^k$ ).
3. Вычислить величины  $g_1(y^k), \dots, g_n(y^k)$ , где значения индекса  $n \leq m$  определяются условиями

$$g_j(y^k) \leq 0, \quad 1 \leq j < n, \quad g_n(y^k) > 0, \quad n \leq m.$$

Выявление первого нарушенного ограничения прерывает испытание в точке  $y^k$ . В случае, когда точка  $y^k$  допустима, т.е. когда  $y^s(x^k) \in Q_{m+1}$ , испытание включает вычисление значений всех функционалов задачи. При этом значение индекса принимается равным величине  $n=m+1$ , а тройка

$$y^s(x^k), \quad n=n(x^k), \quad z^k=g_n(y^s(x^k)), \quad (12.32)$$

является *результатом испытания* в точке  $x^k$ .

4. Если  $n(x^k) > 0$ , то есть  $y^k \in D$ , то определить прообразы  $x^{kl} \in [0, 1]$ ,  $0 \leq l \leq L$ , точки  $y^k$ , и интерпретировать испытание, проведенное в точке  $y^k \in D$ , как проведение испытаний в  $L+1$  точке

$$x^{k0}, x^{k1}, \dots, x^{kL}, \quad (12.33)$$

с одинаковыми результатами

$$n(x^{k0}) = n(x^{k1}) = \dots = n(x^{kL}) = n(x^k),$$

$$g_n(y^0(x^{k0})) = g_n(y^1(x^{k1})) = \dots = g_n(y^L(x^{kL})) = z^k.$$

Проинформировать остальные процессоры о результатах испытания в точке  $y^k$ .

В случае если  $n(x^k) = 0$ , то есть  $y^k \notin D$ , результат испытания относится только к  $s$ -й задаче.

Каждый процессор имеет свою копию программных средств, реализующих вычисление функционалов задачи, и решающее правило алгоритма. Для организации взаимодействия на каждом процессоре создается  $L+1$  очередь, в которые процессоры помещают информацию о выполненных итерациях в виде троек: точка очередной итерации, индекс и значение из (12.32), причем индекс заблокированной точки полагается равным  $-1$ , а значение функции в ней не определено. Связи между процессорами посредством очередей  $Q_{ls}$ , где  $l, 0 \leq l \leq L$ , номер передающего процессора и  $s$ ,  $0 \leq s \leq L$ , номер принимающего процессора, проиллюстрированы на рис. 12.12.

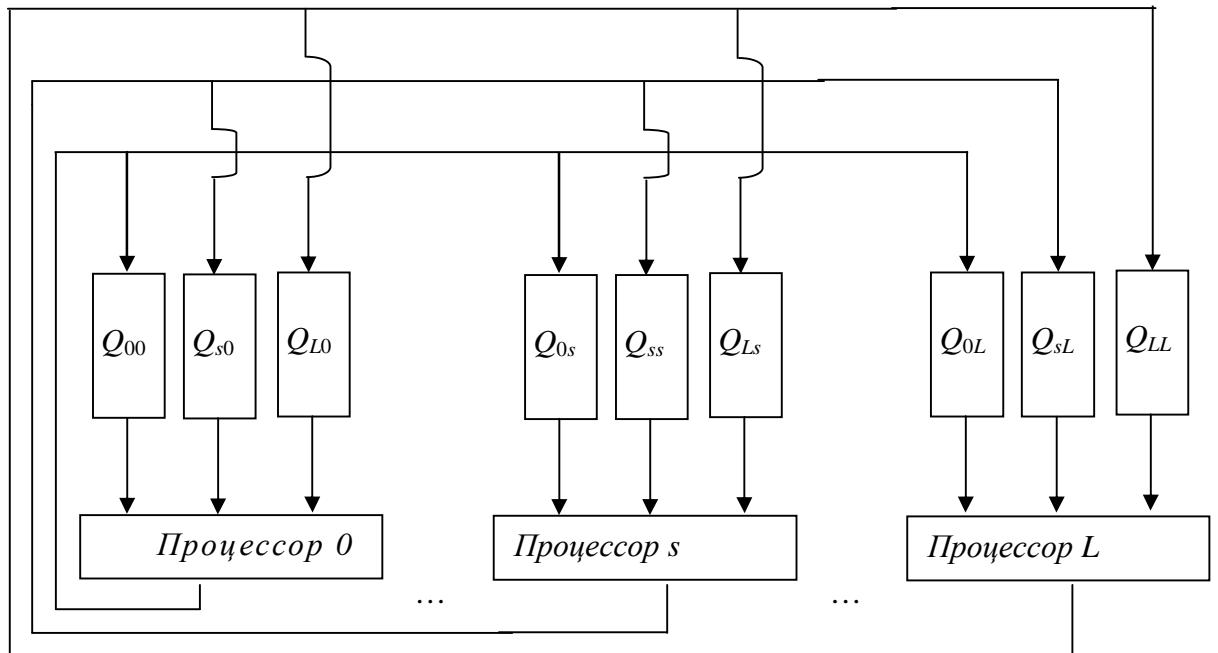


Рис. 12.12. Связи между процессорами

Предлагаемая схема не содержит какого-либо единого управляющего процессора, что увеличивает надежность выполняемых вычислений.

### 12.7.2. Схема алгоритма

Решающие правила предлагаемого параллельного алгоритма в целом совпадают с правилами последовательного алгоритма (кроме способа проведения испытания). Однако для целей более простого освоения материала схема изложена полностью.

Алгоритм выбора точек итераций для всех процессоров одинаков. Начальная итерация осуществляется в заданной точке  $x^1 \in (0, 1)$  (начальные точки для всех процессоров задаются различными). Выбор точки  $x^{q+1}$ ,  $q \geq 1$ , любого последующего испытания определяется следующими правилами.

*Правило 1.* Извлечь из всех очередей, закрепленных за данным процессором, записанные для него результаты, включающие множество  $Y_q = \{y^{qi} : 1 \leq i \leq s_q\}$  точек итераций в области (12.23) и вычисленные в них значения индекса и величин из (12.32). Определить множество  $X_q = \{x^{qi} : 1 \leq i \leq s_q\}$  прообразов точек множества  $Y_k$  при соответствии  $y^l(x)$ .

*Правило 2.* Точки множества итераций

$$\{x_1\} \cup X_1 \cup \dots \cup X_q$$

перенумеровать нижними индексами в порядке увеличения значений координаты

$$0 = x_0 < x_1 < \dots < x_i < \dots < x_k < x_{k+1} = 1, \quad (12.34)$$

где  $k = 1 + s_1 + \dots + s_q$ , и сопоставить им значения  $z_i = g_n(x_i)$ ,  $n = n(x_i)$ ,  $1 \leq i \leq k$ , вычисленные в этих точках. При этом индекс блокированной точки  $x_i$  (т.е. точки, в которой начато проведение испытания другим процессором) полагается равным  $-1$ , т.е.  $n(x_i) = -1$ , значение  $z_i$  является неопределенным. Точки  $x_0$ ,  $x_{k+1}$  введены дополнительно для удобства последующего изложения, индекс данных точек полагается равным  $-2$ , т.е.  $n(x_0) = n(x_{k+1}) = -2$ , а значения  $z_0$ ,  $z_{k+1}$  являются неопределенными.

*Правило 3.* Провести классификацию номеров  $i$ ,  $1 \leq i \leq k$ , точек из ряда (12.34) по числу ограничений задачи, выполняющихся в этих точках, путем построения множеств

$$\begin{aligned} I_{-2} &= \{0, k+1\}, \\ I_{-1} &= \{i : 1 \leq i \leq k, n(x_i) = -1\}, \\ I_n &= \{i : 1 \leq i \leq k, n(x_i) = n\}, \quad 0 \leq n \leq m+1, \end{aligned}$$

содержащих номера всех точек  $x_i$ ,  $1 \leq i \leq k$ , имеющих индексы, равные одному и тому же значению  $n$ . Определить максимальное значение индекса

$$M = \max\{n = n(x_i), 1 \leq i \leq k\}.$$

*Правило 3.* Вычислить текущие нижние границы

$$m_n = \max \left\{ \frac{|z_i - z_j|}{(x_i - x_j)^{1/N}} : j < i, j \in I_n, i \in I_n \right\} \quad (12.35)$$

для относительных разностей функций  $g_n$ ,  $1 \leq n \leq m+1$ . Если множество  $I_n$  содержит менее двух элементов или если  $m_n$  из (12.35) оказывается равным нулю, то принять  $m_n = 1$ . Из (12.35) следует, что оценки  $m_n$  являются неубывающими, начиная с момента, когда (12.35) порождает первое положительное значение  $m_n$ .

*Правило 4.* Для всех непустых множеств  $I_n$ ,  $1 \leq n \leq m+1$ , вычислить оценки

$$z_n^* = \begin{cases} 0, & n < M, \\ \min\{g_n(x_i) : i \in I_n\}, & n = M, \end{cases}$$

*Правило 5.* Для каждого интервала  $(x_{i-1}, x_i)$ ,  $1 \leq i \leq k+1$ , вычислить характеристику  $R(i)$ , где

$$R(i) = \Delta_i + \frac{(z_i - z_{i-1})^2}{r_n^2 m_n^2 \Delta_i} - 2 \frac{(z_i + z_{i-1} - 2z_n^*)}{r_n m_n}, \quad n = n(x_{i-1}) = n(x_i),$$

$$R(i) = 2\Delta_i - 4 \frac{(z_i - z_n^*)}{r_n m_n}, \quad n(x_{i-1}) < n(x_i) = n,$$

$$R(i) = 2\Delta_i - 4 \frac{(z_{i-1} - z_n^*)}{r_n m_n}, \quad n = n(x_{i-1}) > n(x_i),$$

$$\Delta_i = (x_i - x_{i-1})^{1/N}.$$

Величины  $r_n > 1$ ,  $1 \leq n \leq m+1$ , являются параметрами алгоритма.

*Правило 6.* Определить интервал  $(x_{t-1}, x_t)$ , которому соответствует максимальная характеристика

$$R(t) = \max \{ R(i) : 1 \leq i \leq k+1 \}. \quad (12.36)$$

*Правило 7.* Провести очередное испытание в серединной точке интервала  $(x_{t-1}, x_t)$ , если индексы его концевых точек не совпадают, т.е.

$$x^{q+1} = (x_t + x_{t-1})/2, \quad n(x_{t-1}) \neq n(x_t).$$

В противном случае провести испытание в точке

$$x^{q+1} = (x_t + x_{t-1})/2 - \text{sign}(z_t - z_{t-1}) \frac{1}{2r_n} \left[ \frac{|z_t - z_{t-1}|}{m_n} \right]^N, \quad n = n(x_{t-1}) = n(x_t).$$

В случае если  $n(x^{q+1}) = 0$ , т. е.  $y^q \notin D$ , то результаты испытания занести только в очередь, закрепленной за данным процессором на нем самом. Если же  $n(x^{q+1}) > 0$ , т. е.  $y^q \in D$ , то результаты испытания занести во все очереди, закрепленные за данным процессором. Увеличив  $q$  на единицу, перейти к новой итерации.

Описанные правила можно дополнить условием остановки, прекращающим испытания, если

$$\Delta_t \leq \Delta,$$

где  $t$  из (12.36) и  $\Delta > 0$  имеет порядок желаемой покоординатной точности в задаче.

### 12.7.3. Результаты численных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного индексного метода проводились на кластере на базе процессоров Pentium III Xeon 1000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows 2000.

В качестве тестовой задачи рассматривалась задача из примера 12.1: минимизировать функцию

$$\begin{aligned} j(y_1, y_2) = & -1.5y_1^2 \exp\{1 - y_1^2 - 20.25(y_1 - y_2)^2\} - \\ & - [0.5(y_1 - 1)(y_2 - 1)]^4 \exp\{2 - [0.5(y_1 - 1)]^4 - (y_2 - 1)^4\} \end{aligned}$$

в области поиска  $0 \leq y_1 \leq 4$ ,  $-1 \leq y_2 \leq 3$ , при ограничениях

$$g_1(y_1, y_2) = 0.01[(y_1 - 2.2)^2 + (y_2 - 1.2)^2 - 2.25] \leq 0$$

$$g_2(y_1, y_2) = 100[1 - (y_1 - 2)^2 / 1.44 - (0.5y_2)^2] \leq 0$$

$$g_3(y_1, y_2) = 10[y_2 - 1.5 - 1.5 \sin(6.283(y_1 - 1.75))] \leq 0$$

Допустимые по первому ограничению точки образуют круг с границей  $g_1(y_1, y_2) = 0$ . Допустимые по второму ограничению точки находятся во внешности эллипса  $g_2(y_1, y_2) = 0$ . Точки, допустимые по третьему ограничению, находятся ниже синусоиды  $g_3(y_1, y_2) = 0$ . Глобальный минимум  $j(y_1^*, y_2^*) = -1.489$  достигается в точке  $(y_1^*, y_2^*) = (0.942, 0.944)$  (см. рис. 12.13).

Данная задача была решена с использованием параллельного индексного алгоритма со следующими параметрами: параметр надежности  $r_1 = \dots = r_4 = 1.4$ , точность решения  $e = 10^{-3}$ , плотность построения развертки  $m = 12$ . Число процессоров и, соответственно, число разверток, варьировалось от 2 до 6. Во всех экспериментах была получена одна и та же оценка оптимума  $(y_1^*, y_2^*) = (0.942, 0.945)$ .

На рисунке 12.13 изображена область поиска – квадрат, где выделена допустимая область задачи и обозначены точки испытаний, полученные при применении параллельного индексного метода с шестью развертками.

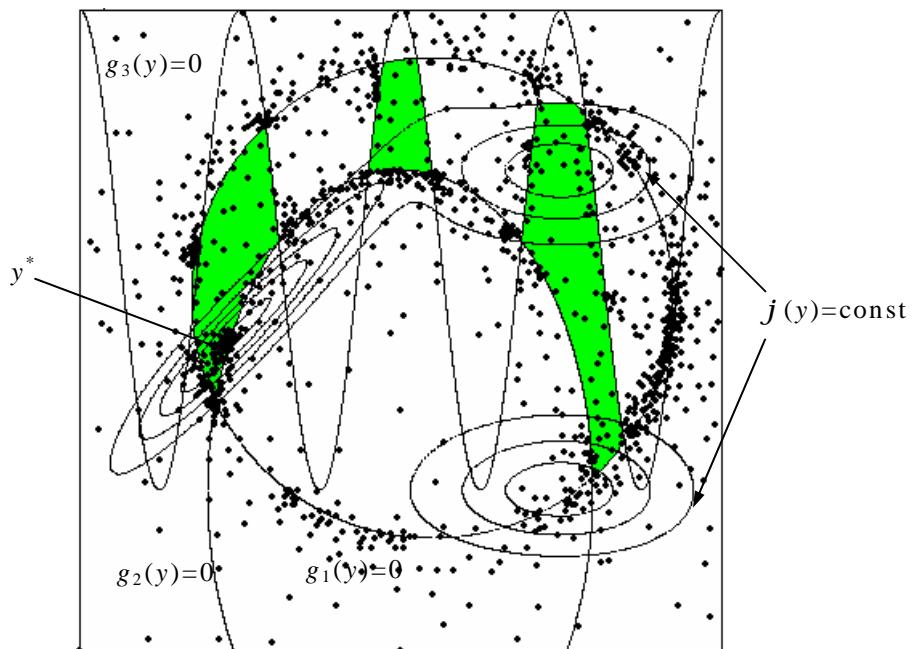


Рис. 12.13. Результаты решения задачи с использованием шести процессоров

В табл. 12.2–12.4 приведены подробные результаты экспериментов (табл. 12.1 соответствует последовательному алгоритму и приведена для сравнения).

Табл. 12.1. Результаты решения задачи с использованием одного процессора

$l$	Вычислено значений функции				
	$g_1$	$g_2$	$g_3$	$j = g_4$	
1	1098	623	392	152	

Табл. 12.2. Результаты решения задачи с использованием двух процессоров

$l$	Вычислено значений функции				
	$g_0$	$g_1$	$g_2$	$g_3$	$j = g_4$
1	636	567	331	212	104
2	632	327	193	125	61

Всего	1268	894	524	337	165
-------	------	-----	-----	-----	-----

Табл. 12.3. Результаты решения задачи с использованием четырех процессоров

$l$	Вычислено значений функции				
	$g_0$	$g_1$	$g_2$	$g_3$	$j = g_4$
1	585	508	311	206	88
2	579	262	157	86	36
3	579	293	174	119	41
4	579	284	178	116	48
Всего	2322	1347	820	527	213

Табл. 12.4. Результаты решения задачи с использованием шести процессоров

$l$	Вычислено значений функции				
	$g_0$	$g_1$	$g_2$	$g_3$	$j = g_4$
1	354	289	165	105	44
2	346	90	55	28	10
3	346	113	66	40	10
4	346	160	87	48	25
5	346	224	119	68	23
6	346	274	159	97	37
Всего	2084	1150	651	386	149

В таблице 12.5 приведена оценка ускорения, возникающего при использовании параллельного алгоритма. Ускорение определялось как отношение числа итераций (которое совпадает с числом проверок первого ограничения  $g_1$ ), выполненных последовательным алгоритмом, и наибольшего числа итераций, выполненных параллельным алгоритмом на одном из процессоров. Данный способ определения ускорения актуален для задач, в которых оценка функционалов задачи требует заметных вычислительных ресурсов.

Табл. 12.5. Результаты численных экспериментов

Число процессоров	Ускорение
1	—
2	1.93
4	2.16
6	3.8

Следует отметить, что результаты экспериментов демонстрируют (см. рис. 12.13) концентрацию точек испытаний не только в окрестности точки глобального оптимума, но и в окрестностях линий нулевого уровня ограничений (т.е. граничных точек допустимой области). Это обусловлено тем, что граничные точки допустимой области являются локально-оптимальными для индексной функции (12.13). Концентрация точек испытаний вне окрестности глобального оптимума замедляет сходимость метода при решении многомерных задач.

В рамках разработанной теории предложены способы ускорения сходимости индексного алгоритма, основанные на учете степени регулярности задачи, определяемой на основе введенного понятия  $\epsilon$ -резервированных решений, а также путем использования адаптивно изменяемого порядка проверки ограничений. Рассмотрение модификаций индексного алгоритма выходит за рамки данного учебного материала, желающие могут их

найти в работах [40,97]. Ниже мы лишь приведем результаты решения тестового примера с использованием ускоренного параллельного индексного алгоритма.

Табл. 12.6. Результаты решения задачи с использованием шести процессоров

$l$	Вычислено значений функции				
	$g_0$	$g_1$	$g_2$	$g_3$	$j = g_4$
1	122	102	57	36	18
2	114	48	30	18	10
3	114	44	31	21	10
4	114	49	33	15	11
5	114	56	28	11	14
6	114	66	49	32	17
Всего	692	365	228	133	80

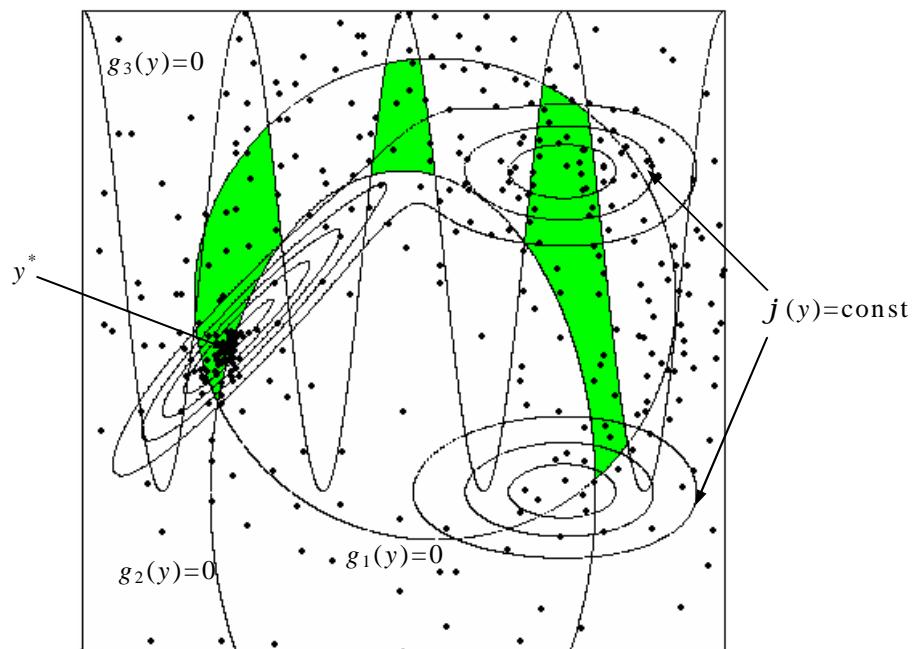


Рис. 12.14. Результаты решения задачи с использованием шести процессоров

Приведенные на рис. 14.14 результаты решения задачи наглядно демонстрируют эффект снижения концентрации точек итераций вне допустимой области.

Глава 13. Программная система Паралаб для изучения и исследования методов параллельных вычислений .....	2
13.1. Введение.....	2
13.2. Общая характеристика системы .....	3
13.3. Формирование модели вычислительной системы .....	5
13.3.1. Выбор топологии сети .....	5
13.3.2. Задание количества процессоров и ядер.....	7
13.3.3. Задание характеристик сети .....	8
13.4. Постановка вычислительной задачи и выбор параллельного метода решения .....	10
13.4.1. Умножение матрицы на вектор .....	12
13.4.2. Матричное умножение .....	14
13.4.3. Решение систем линейных уравнений .....	16
13.4.4. Сортировка данных .....	18
13.4.5. Обработка графов.....	21
13.4.6. Решение дифференциальных уравнений .....	24
13.4.7. Решение задач многоэкстремальной оптимизации.....	27
13.5. Определение графических форм наблюдения за процессом параллельных вычислений.....	29
13.5.1. Область "Выполнение эксперимента" .....	31
13.5.2. Область "Текущее состояние массива" .....	33
13.5.3. Область "Результат умножения матрицы на вектор" .....	33
13.5.4. Область "Результат умножения матриц" .....	34
13.5.5. Область "Результат решения системы уравнений".....	34
13.5.6. Область "Результат обработки графа".....	34
13.5.7. Область "Распределение тепла" .....	34
13.5.8. Область "Целевая функция" .....	35
13.5.9. Выбор процессора .....	35
13.6. Накопление и анализ результатов экспериментов .....	35
13.6.1. Общие результаты экспериментов.....	36
13.6.2. Просмотр итогов .....	36
13.7. Выполнение вычислительных экспериментов .....	38
13.7.1. Последовательное выполнение экспериментов .....	38
13.7.2. Выполнение экспериментов по шагам .....	39
13.7.3. Выполнение нескольких экспериментов .....	39
13.7.4. Выполнение серии экспериментов.....	41
13.7.5. Выполнение реальных вычислительных экспериментов .....	42
13.8. Использование результатов экспериментов: запоминание, печать и перенос в другие программы .....	43
13.8.1. Запоминание результатов.....	43
13.8.2. Печать результатов экспериментов.....	44
13.8.3. Копирование результатов в другие программы .....	44
Приложение: Таблица выполнимости методов на различных топологиях в системе Паралаб.....	45

## Глава 13.

# Программная система Параллаб для изучения и исследования методов параллельных вычислений

### 13.1. Введение

Программная система **Параллельная Лаборатория** (сокращенное наименование **Параллаб**) обеспечивает возможность проведения вычислительных экспериментов с целью изучения и исследования параллельных алгоритмов решения сложных вычислительных задач. Система может быть использована для организации лабораторного практикума по различным учебным курсам в области параллельного программирования, в рамках которого обеспечивается возможность

- моделирования многопроцессорных и многоядерных вычислительных систем с различной топологией сети передачи данных;
- получения визуального представления о вычислительных процессах и операциях передачи данных, происходящих при параллельном решении разных вычислительных задач;
- построения оценок эффективности изучаемых методов параллельных вычислений.

Проведение такого практикума может быть организовано на "обычных" однопроцессорных компьютерах, работающих под управлением операционных систем MS Windows (режим многозадачной имитации параллельных вычислений). Кроме режима имитации, в системе Параллаб обеспечивается удаленный доступ к имеющейся многопроцессорной вычислительной системе для выполнения экспериментов в режиме "настоящих" параллельных вычислений для сопоставления результатов имитации и реальных расчетов.

В целом система Параллаб представляет собой *интегрированную среду для изучения и исследования параллельных алгоритмов* решения сложных вычислительных задач. Широкий набор имеющихся средств визуализации процесса выполнения эксперимента и анализа полученных результатов позволяет изучить эффективность использования тех или иных алгоритмов на разных вычислительных системах, сделать выводы о масштабируемости алгоритмов и определить возможное ускорение процесса параллельных вычислений.

Реализуемые системой Параллаб процессы изучения и исследований ориентированы на активное освоение основных теоретических положений и способствуют формированию у пользователей своих собственных представлений о моделях и методах параллельных вычислений путем наблюдения, сравнения и сопоставления широкого набора различных визуальных графических форм, демонстрируемых в ходе выполнения вычислительного эксперимента.

Основной сферой использования системы Параллаб является *учебное применение* студентами и преподавателями вузов для исследований и изучения параллельных алгоритмов решения сложных вычислительных задач в рамках лабораторного практикума по различным учебным курсам в области параллельного программирования. Система Параллаб может использоваться также и при проведении *научных исследований* для оценки эффективности параллельных вычислений.

Пользователи, начинающие знакомиться с проблематикой параллельных вычислений, найдут систему Паралаб полезной для освоения методов параллельного программирования, опытные вычислители могут использовать систему для оценки эффективности новых разрабатываемых параллельных алгоритмов.

## 13.2. Общая характеристика системы

Паралаб - программная система, которая позволяет проводить как реальные параллельные вычисления на многопроцессорной вычислительной системе, так и имитировать такие эксперименты на одном последовательном компьютере с визуализацией процесса параллельного решения сложной вычислительной задачи.

При проведении имитационных экспериментов Паралаб предоставляет возможность для пользователя:

- определить топологию параллельной вычислительной системы для проведения экспериментов, задать число процессоров и ядер в этой топологии, установить производительность процессоров, выбрать характеристики коммуникационной среды и способ коммуникации;
- осуществить постановку вычислительной задачи, для которой в составе системы Паралаб имеются реализованные параллельные алгоритмы решения, выполнить задание параметров задачи;
- выбрать параллельный метод для решения выбранной задачи;
- установить параметры визуализации для выбора желаемого темпа демонстрации, способа отображения пересылаемых между процессорами данных, степени детальности визуализации выполняемых параллельных вычислений;
- выполнить эксперимент для параллельного решения выбранной задачи; при этом в системе Паралаб может быть сформировано несколько различных заданий для проведения экспериментов с отличающимися типами многопроцессорных систем, задач или методов параллельных вычислений, для которых выполнение эксперимента может происходить одновременно (в режиме разделения времени); одновременное выполнение эксперимента для нескольких заданий позволяет наглядно сравнивать динамику решения задачи различными методами, на разных топологиях, с разными параметрами исходной задачи. При выполнении серии экспериментов, требующих длительных вычислений, в системе имеется возможность их проведения в автоматическом режиме с запоминанием результатов в журнале экспериментов для организации последующего анализа полученных данных;
- накапливать и анализировать результаты выполненных экспериментов; по запомненным результатам в системе имеется возможность построения графиков, характеризующих параллельные вычисления зависимостей (времени решения, ускорения) от параметров задачи и вычислительной системы.

Одной из важнейших характеристик системы является возможность выбора способов проведения экспериментов. Эксперимент может быть выполнен в *режиме имитации*, т.е. проведен на одном процессоре без использования каких-либо специальных программных средств типа библиотек передачи сообщений. Кроме того, в рамках системы Паралаб обеспечивается возможность следующих способов проведения *реального вычислительного эксперимента*:

- на одном компьютере, где имеется библиотека передачи сообщений MPI (выполнение эксперимента в режиме разделения времени); для данной библиотеки имеются общедоступные реализации, которые могут быть получены в сети Интернет и установлены на компьютере под управлением операционных систем MS Windows,

- на реальной многопроцессорной кластерной вычислительной системе,
- в режиме удаленного доступа к вычислительному кластеру.

При построении зависимостей временных характеристик от параметров задачи и вычислительной системы для экспериментов, выполненных в режиме имитации, используются теоретические оценки в соответствии с имеющимися моделями параллельных вычислений (см. например, [8,10,67]). Для реальных экспериментов на многопроцессорных вычислительных системах зависимости строятся по набору результатов проведенных вычислительных экспериментов.

Важно отметить, что в системе ПараЛаб обеспечена возможность ведения журнала экспериментов для запоминания результатов выполненных экспериментов. Запомненные результаты позволяют выполнить анализ полученных данных; по имеющейся в журнале информации любой из проведенных ранее экспериментов может быть восстановлен для повторного выполнения или продолжения расчетов.

Реализованные таким образом процессы изучения и исследований позволяют освоить теоретические положения и помогут формированию представлений о методах построения параллельных алгоритмов, ориентированных на решение конкретных прикладных задач.

### ***Демонстрационный пример***

Для выполнения примера, имеющегося в комплекте поставки системы:

- выберите пункт меню **Начало** и выполните команду **Загрузить**;
- выберите строку **first.prl** в списке имен файлов и нажмите кнопку **Открыть**;
- выберите пункт меню **Выполнение** и выполните команду **В активном окне**.

В результате выполненных действий на экране дисплея будет представлено окно для выполнения вычислительного эксперимента (рис. 13.1). В этом окне демонстрируется решение задачи умножения матриц при помощи ленточного алгоритма.

В области "Выполнение эксперимента" представлены данные процессоров, которые они обрабатывают в каждый момент выполнения алгоритма (для ленточного алгоритма умножения матриц – это несколько последовательных строк матрицы  $A$  и несколько последовательных столбцов матрицы  $B$ ), а так же структура линий коммутации. При помощи динамически перемещающихся прямоугольников (пакетов) желтого цвета изображается обмен данными, который осуществляют процессоры.

В области "Результат умножения матриц" изображается текущее состояние матрицы – результата умножения. Поскольку результатом перемножения полос исходных матриц  $A$  и  $B$  является блок матрицы  $C$ , получаемая результирующая матрица имеет блочную структуру. Темно-синим цветом обозначены уже вычисленные блоки, голубым цветом выделены блоки, еще подлежащие определению.

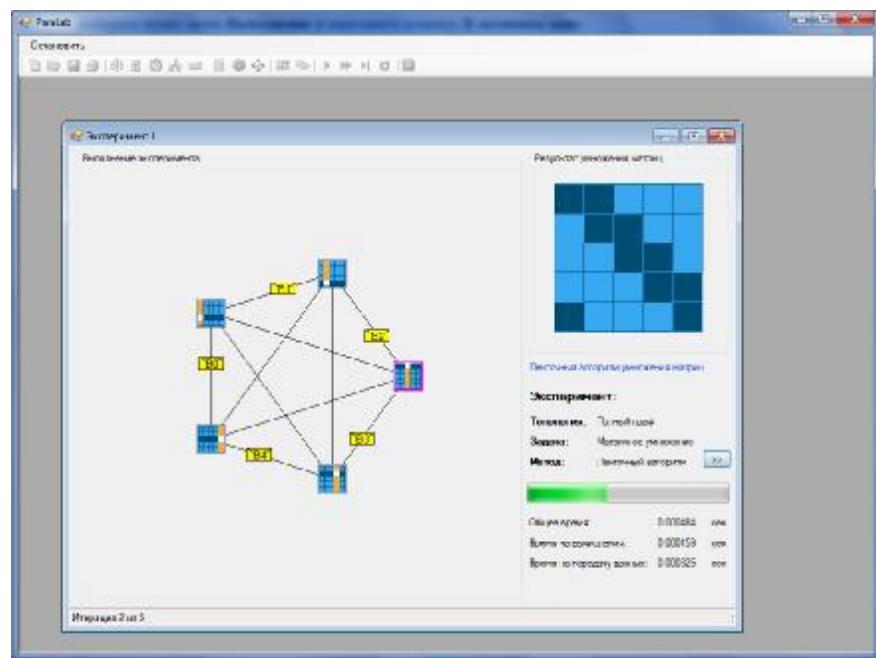


Рис. 13.1. Окно вычислительного эксперимента

В области "Эксперимент" отображены основные параметры выполняемого эксперимента. К таким параметрам относятся топология, решаемая задача, а также метод ее решения. Ленточный индикатор области "Эксперимент" отображает текущую стадию выполнения алгоритма. В строках "Общее время", "Время на вычисления" и "Время на передачу данных" представлены временные характеристики алгоритма.

После выполнения эксперимента (восстанавливается главное меню системы) можно завершить работу системы. Для этого выберите пункт меню **Архив** и выполните команду **Выход**.

### 13.3. Формирование модели вычислительной системы

Для формирования модели вычислительной системы необходимо определить топологию сети, количество процессоров и ядер на них, производительность каждого процессора и характеристики коммуникационной среды (латентность, пропускную способность и метод передачи данных). Следует отметить, что в рамках системы Паралаб вычислительная система полагается однородной, т.е. все процессоры обладают одинаковой производительностью, а все каналы связи имеют одинаковые характеристики.

#### 13.3.1. Выбор топологии сети

*Топология сети передачи данных* определяется структурой линий коммутации между процессорами вычислительной системы. В системе Паралаб обеспечивается поддержка следующих типовых топологий:

- **полный граф** (*completely-connected graph or clique*) – система, в которой между любой парой процессоров существует прямая линия связи; как результат, данная топология обеспечивает минимальные затраты при передаче данных, однако является сложно реализуемой при большом количестве процессоров;
- **линейка** (*linear array or farm*) – система, в которой каждый процессор имеет линии связи только с двумя соседними (с предыдущим и последующим) процессорами; такая схема является, с одной стороны, просто реализуемой, а с другой стороны, соответствует структуре передачи данных при решении многих вычислительных задач (например, при организации конвейерных вычислений);

- **кольцо** (*ring*) – данная топология получается из линейки процессоров соединением первого и последнего процессоров линейки;
- **звезда** (*star*) – система, в которой выделенный центральный процессор, у которого с любым процессором существует прямая линия связи; все остальные процессы имеют связь только с центральным; как результат, данная топология обеспечивает малые затраты при реализации, но при большом количестве процессоров вызывает очень большую нагрузку на центральный процессор;
- **решетка** (*mesh*) – система, в которой граф линий связи образует прямоугольную двухмерную сетку; подобная топология может быть достаточно просто реализована и, кроме того, может быть эффективно используема при параллельном выполнении многих численных алгоритмов (например, при реализации методов блочного умножения матриц);
- **гиперкуб** (*hypercube*) – данная топология представляет частный случай структуры  $N$ -мерной решетки, когда по каждой размерности сетки имеется только два процессора (т.е. гиперкуб содержит  $2^N$  процессоров при размерности  $N$ ); данный вариант организации сети передачи данных достаточно широко распространен в практике и характеризуется следующим рядом отличительных признаков:
  - два процессора имеют соединение, если двоичное представление их номеров имеет только одну различающуюся позицию;
  - в  $N$ -мерном гиперкубе каждый процессор связан ровно с  $N$  соседями;
  - $N$ -мерный гиперкуб может быть разделен на два  $(N-1)$ -мерных гиперкуба (всего возможно  $N$  различных таких разбиений);
  - кратчайший путь между двумя любыми процессорами имеет длину, совпадающую с количеством различающихся битовых значений в номерах процессоров (данная величина известна как *расстояние Хэмминга*).

### ***Правила использования системы ПароЛаб***

**1. Запуск системы.** Для запуска системы ПароЛаб выделите пиктограмму системы и выполните двойной щелчок левой кнопкой мыши (или нажмите клавишу **Enter**). Далее выполните команду **Выполнить новый эксперимент** (пункт меню **Начало**) и нажмите в диалоговом окне **Название эксперимента** кнопку **OK** (при желании до нажатия кнопки **OK** может быть изменено название создаваемого окна для проведения экспериментов).

**2. Выбор топологии вычислительной системы.** Для выбора топологии вычислительной системы следует выполнить команду **Топология** пункта меню **Система**. В появившемся диалоговом окне (рис. 13.2) щелкните левой клавишей мыши на пиктограмме нужной топологии или внизу в области соответствующей круглой кнопки выбора (радиокнопки). При нажатии кнопки **Справка** можно получить справочную информацию о реализованных топологиях. Нажмите кнопку **OK** для подтверждения выбора и кнопку **Отмена** для возврата в основное меню системы ПароЛаб.

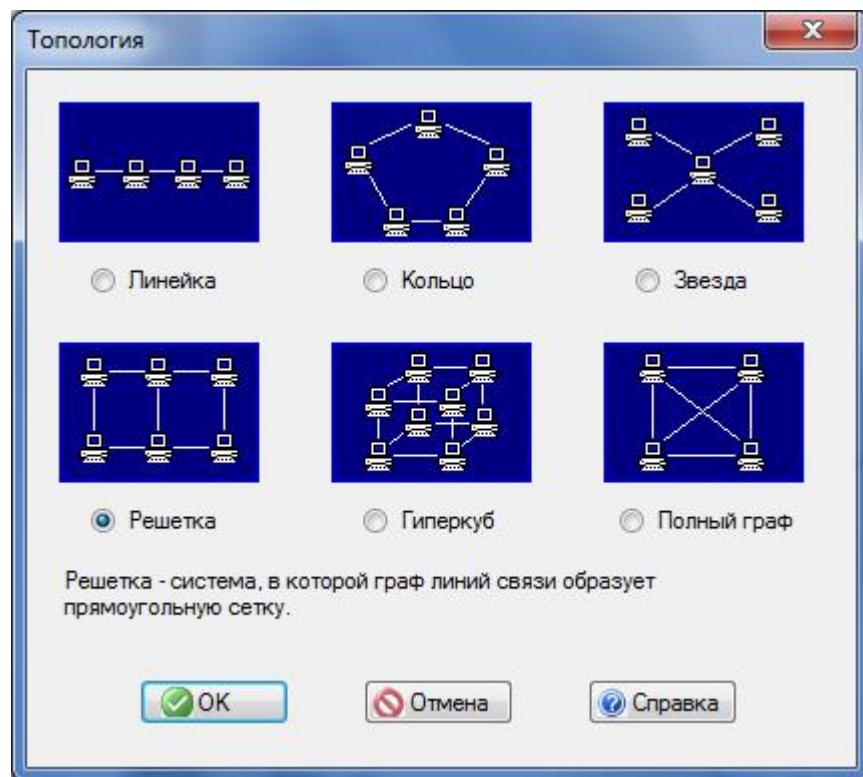


Рис. 13.2. Диалоговое окно для выбора топологии

### 13.3.2. Задание количества процессоров и ядер

Для выбранной топологии системы Паралаб позволяет установить необходимое количество процессоров и ядер. Выполняемый при этом выбор конфигурации системы осуществляется в соответствии с типом используемой топологии. Так, например, число процессоров в двухмерной решетке должно являться полным квадратом (размеры решетки по горизонтали и вертикали совпадают), а число процессоров в гиперкубе – степенью числа 2. В данной реализации системы ядер на процессоре может быть одно, два или четыре.

Под *производительностью процессора (каждого ядра)* в системе Паралаб понимается количество операций с плавающей запятой, которое процессор может выполнить за секунду (floating point operations per second – flops). Важно отметить, что при построении оценок времени выполнения эксперимента предполагается, что все машинные команды являются одинаковыми и соответствуют одной и той же операции с плавающей точкой.

#### *Правила использования системы Паралаб*

**1. Задание количества процессоров и ядер.** Для выбора числа процессоров и ядер на процессоре необходимо выполнить команду **Количество Процессоров** пункта меню **Система**. В появившемся диалоговом окне (рис. 13.3) Вам предоставляется несколько пиктограмм со схематическим изображением числа узлов вычислительной системы. Для каждого узла есть возможность задать количество процессоров и ядер на процессоре. Для выбора щелкните левой клавишей мыши на нужной пиктограмме. Пиктограмма, соответствующая текущему числу узлов, выделена ярко-синим цветом.

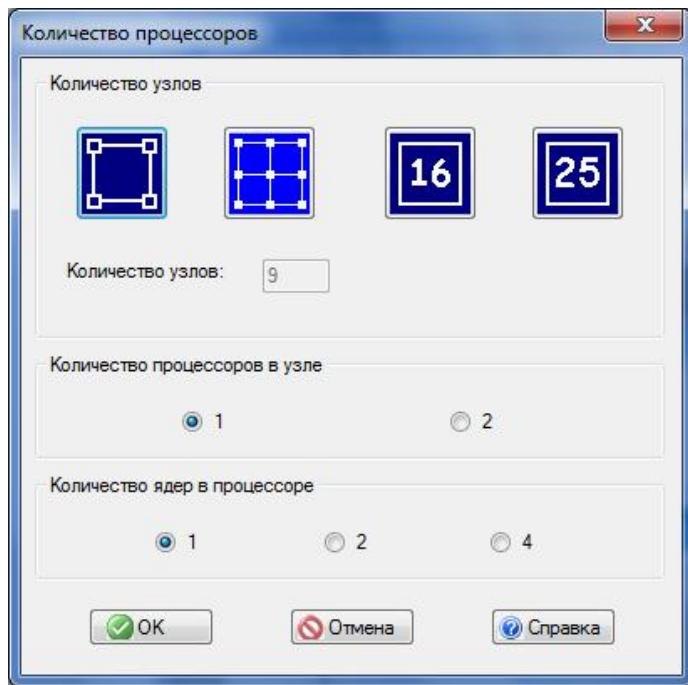


Рис. 13.3. Диалоговое окно для задания числа процессоров

Нажмите кнопку **OK** для подтверждения выбора или кнопку **Отмена** для возврата в основное меню системы Паралаб без изменения числа процессоров.

**2. Определение производительности ядер процессора.** Для задания производительности ядер процессоров, составляющих многопроцессорную вычислительную систему, следует выполнить команду **Производительность Процессоров** пункта меню **Система**. Далее в появившемся диалоговом окне (рис. 13.4) при помощи бегунка или поля ввода задать величину производительности. Для подтверждения выбора нажмите кнопку **OK** (или клавишу **Enter**). Для возврата в основное меню системы Паралаб без изменений нажмите кнопку **Отмена** (или клавишу **Escape**).

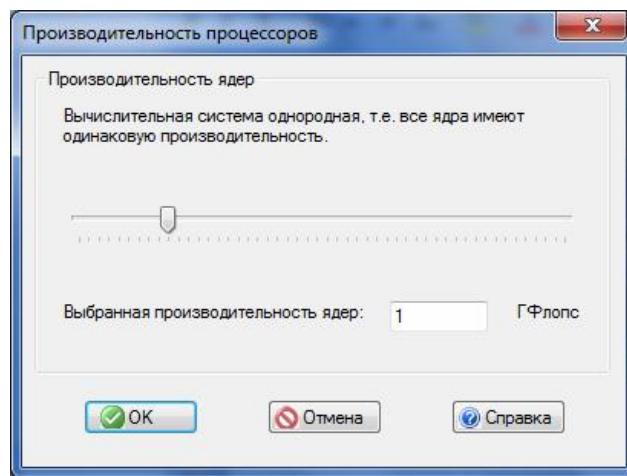


Рис. 13.4. Диалоговое окно для задания производительности процессора

### 13.3.3. Задание характеристик сети

Время передачи данных между процессорами определяет коммуникационную составляющую (*communication overhead*) длительности выполнения параллельного алгоритма в многопроцессорной вычислительной системе. Основной набор параметров, описывающих время передачи данных, состоит из следующего ряда величин:

– *латентность (a)* - время начальной подготовки, которое характеризует длительность подготовки сообщения для передачи, поиска маршрута в сети и т.п.;

– *пропускная способность сети (b)* – определяется как максимальный объем данных, который может быть передан за некоторую единицу времени по одному каналу передачи данных. Данная характеристика измеряется, например, количеством переданных бит в секунду.

К числу реализованных в системе Паралаб методов передачи данных относятся следующие два широко известных способа коммуникации (см. [10,72]). Первый из них ориентирован на *передачу сообщений* (МПС) как неделимых (атомарных) блоков информации (*store-and-forward routing or SFR*). При таком подходе процессор, содержащий исходное сообщение, готовит весь объем данных для передачи, определяет транзитный процессор, через который данные могут быть доставлены целевому процессору, и запускает операцию пересылки данных. Процессор, которому направлено сообщение, в первую очередь осуществляет прием полностью всех пересылаемых данных и только затем приступает к пересылке принятого сообщения далее по маршруту. Время пересылки данных  $T$  для метода передачи сообщения размером  $m$  по маршруту длиной  $l$  определяется выражением:

$$T = (a + (m/b)) \cdot l.$$

Второй способ коммуникации основывается на представлении пересылаемых сообщений в виде блоков информации меньшего размера (*пакетов*), в результате чего передача данных может быть сведена к *передаче пакетов* (МПП). При таком методе коммуникации (*cut-through routing or CTR*) транзитный процессор может осуществлять пересылку данных по дальнейшему маршруту непосредственно сразу после приема очередного пакета, не дожидаясь завершения приема данных всего сообщения. Количество передаваемых при этом пакетов равно

$$n = \left\lceil \frac{m}{V - V_0} \right\rceil + 1,$$

где  $V$  есть размер пакета, а величина  $V_0$  определяет объем служебных данных, передаваемых в каждом пакете (*заголовок пакета*). Как результат, время передачи сообщения в этом случае составит

$$T = a + \frac{V}{b} \left( l + \left\lceil \frac{m}{V - V_0} \right\rceil \right) = a + \frac{V}{b} (l + n - 1)$$

(скобки  $\lceil \rceil$  обозначают операцию приведения к целому с избытком).

Сравнивая полученные выражения, можно заметить, что в случае, когда длина маршрута больше единицы, метод передачи пакетов приводит к более быстрой пересылке данных; кроме того, данный подход снижает потребность в памяти для хранения пересылаемых данных для организации приема-передачи сообщений, а для передачи пакетов могут использоваться одновременно разные коммуникационные каналы.

### **Правила использования системы Паралаб**

**1. Определение характеристик коммуникационной среды.** Для определения характеристик сети выполните команду **Характеристики сети** пункта меню **Система**. В открывшемся диалоговом окне (рис. 13.5) при помощи бегунков можно задать время начальной подготовки данных (латентность) в микросекундах и пропускную способность каналов сети (Мбайт/с). Для подтверждения выбора нажмите кнопку **OK**. Для возврата в основное меню системы Паралаб без изменения этих параметров нажмите кнопку

**Отмена.**

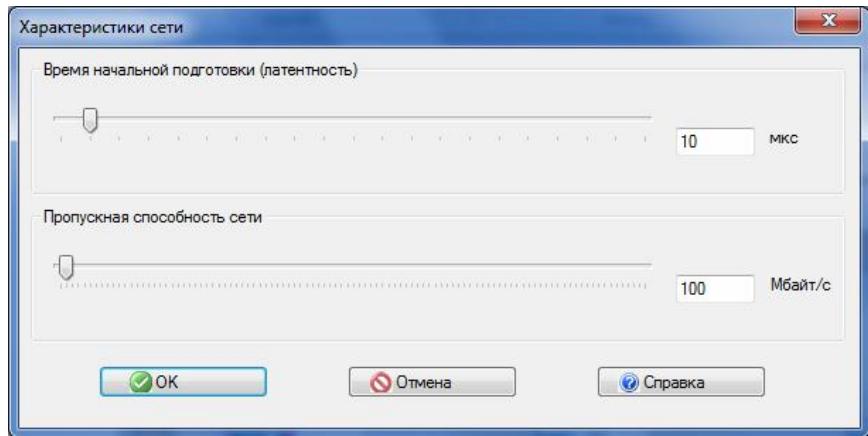


Рис. 13.5. Диалоговое окно для задания характеристик сети

**2. Определение метода передачи данных.** Для определения метода передачи данных, который будет использоваться при проведении вычислительного эксперимента и при построении временных характеристик, необходимо выполнить команду **Метод Передачи Данных** пункта меню **Система**. В открывшемся диалоговом окне (рис. 13.6) следует щелкнуть левой клавишей мыши в области радиокнопки, которая соответствует желаемому методу передачи данных. Если выбран метод передачи пакетов, при помощи бегунков возможно задать длину пакета и длину заголовка пакета в байтах. Для подтверждения выбора метода передачи данных и его параметров нажмите кнопку **OK**.

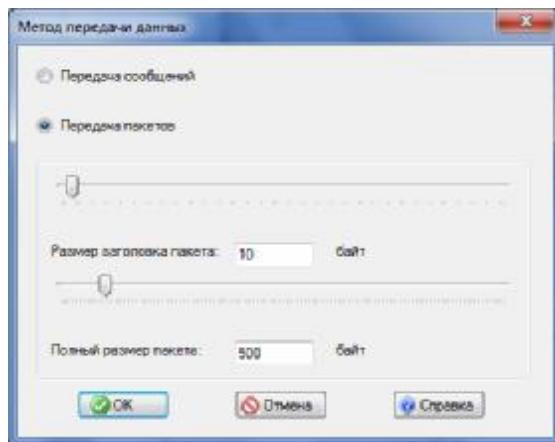


Рис. 13.6. Диалоговое окно для задания метода передачи данных

**3. Завершение работы системы.** Для завершения работы системы Паралаб следует выполнить команду **Завершить** (пункт меню **Архив**).

#### 13.4. Постановка вычислительной задачи и выбор параллельного метода решения

Для параллельного решения тех или иных вычислительных задач процесс вычислений должен быть представлен в виде набора независимых вычислительных процедур, допускающих выполнение на независимых процессорах.

Общая схема организации таких вычислений может быть представлена следующим образом:

- разделение процесса вычислений на части, которые могут быть выполнены одновременно,
- распределение вычислений по процессорам,

- обеспечение взаимодействия параллельно выполняемых вычислений.

Возможные способы получения методов параллельных вычислений:

- разработка новых параллельных алгоритмов,
- распараллеливание последовательных алгоритмов.

Условия эффективности параллельных алгоритмов:

- равномерная загрузка процессоров (отсутствие простоев),
- низкая интенсивность взаимодействия процессоров (независимость).

В системе Паралаб реализованы широко применяемые параллельные алгоритмы для решения ряда сложных вычислительных задач из разных областей научно-технических приложений: алгоритмы сортировки данных, матричных операций, решения систем линейных уравнений, решения дифференциальных уравнений, обработки графов и многоэкстремальной оптимизации.

### *Правила использования системы Паралаб*

**1. Выбор задачи.** Для выбора задачи из числа реализованных в системе выберите пункт меню **Задача** и выделите левой клавишей мыши одну из строк: **Умножение матрицы на вектор**, **Матричное умножение**, **Решение систем лин. уравнений**, **Сортировка**, **Обработка графов**, **Решение диффер. уравнений**, **Многоэкстремальная оптимизация**. Выбранная задача станет текущей в активном окне.

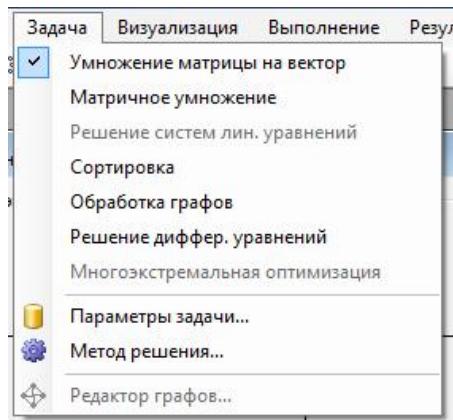


Рис. 13.7. Выбор задачи

Следует отметить, что решение отдельных классов задач может быть выполнено только при использовании вполне определенных топологиях вычислительной системы (так, например, решение задач многоэкстремальной оптимизации возможно только при топологии полного графа). Сводка соответствия классов задач и топологий приводится в приложении к данной главе; см. также файле *MethodsTopologies* в каталоге *Doc* комплекта поставки системы Паралаб.

**2. Определение параметров задачи.** Основным параметром задачи в системе Паралаб является объем исходных данных. Для задачи сортировки – это размер упорядочиваемого массива данных, для матричных операций и задачи решения систем линейных уравнений – размерность исходных матриц, для задачи обработки графов – число вершин в графе. Для выбора параметров задачи необходимо выполнить команду **Параметры задачи** пункта меню **Задача**. В появившемся диалоговом окне (рис. 13.8) следует при помощи бегунка задать необходимый объем исходных данных. Нажмите **OK** (Enter) для подтверждения задания параметра. Для возврата в основное меню системы Паралаб без сохранения изменений нажмите **Отмена** (Escape).

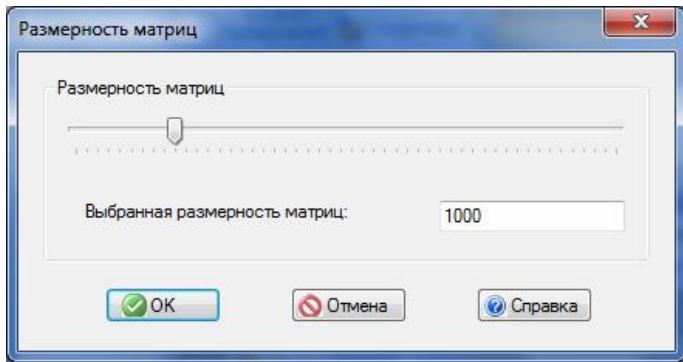


Рис. 13.8. Диалоговое окно задания параметров задачи в случае решения задачи матричного умножения

**3. Определение метода решения задачи.** Для выбора метода решения задачи выполните команду **Метод решения** пункта меню **Задача**. В появившемся диалоговом окне (рис. 13.9) выделите мышью нужный метод, нажмите **OK** для подтверждения выбора, нажмите **Отмена** для возврата в основное меню системы ПароЛаб.

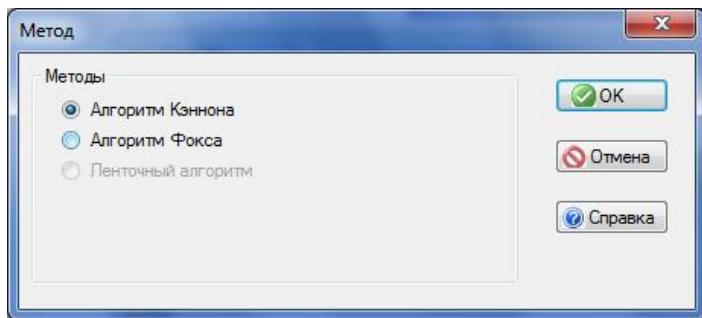


Рис. 13.9. Диалоговое окно выбора метода в случае решения задачи матричного умножения.

### 13.4.1. Умножение матрицы на вектор

В результате *умножения матрицы  $A$  размерности  $m \times n$  и вектора  $b$* , состоящего из  $n$  элементов, получается вектор  $c$  размера  $m$ , каждый  $i$ -ый элемент которого есть результат скалярного умножения  $i$ -й строки матрицы  $A$  (обозначим эту строчку  $a_i$ ) и вектора  $b$ :

$$c_i = (a_i, b) = \sum_{j=1}^n a_{ij} b_j, 1 \leq i \leq m.$$

Тем самым получение результирующего вектора  $c$  предполагает повторение  $m$  однотипных операций по умножению строк матрицы  $A$  и вектора  $b$ . Каждая такая операция включает умножение элементов строки матрицы и вектора  $b$  ( $n$  операций) и последующее суммирование полученных произведений ( $n-1$  операций). Общее количество необходимых скалярных операций есть величина

$$T_1 = m \cdot (2n - 1).$$

Для операции умножения матрицы на вектор характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данний момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем

распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

Более полная информация об алгоритмах матрично-векторного умножения, реализованных в системе Паралаб, содержится в Главе 6.

**1. Умножение матрицы на вектор при разделении данных по строкам.** Данный алгоритм основан на представлении матрицы непрерывными наборами (горизонтальными полосами) строк. Полученные полосы распределяются по процессорам вычислительной системы. Вектор  $b$  копируется на все процессоры. Перемножение полосы матрицы на вектор (а выполнение процессорами этой операции может быть выполнено параллельно) приводит к получению блока элементов результирующего вектора  $c$ . Для объединения результатов расчета и получения полного вектора  $c$  на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных.

### **Задания и упражнения**

1. Создайте в системе Паралаб новое окно вычислительного эксперимента. Для этого окна выберите задачу умножения матрицы на вектор (щелкните левой кнопкой мыши на строке **Умножение матрицы на вектор** пункта меню **Задача**).
2. Откройте диалоговое окно выбора топологии и выберете топологию полный граф.
3. Откройте диалоговое окно выбора метода и убедитесь в том, что выбран метод, основанный на горизонтальном разбиении матрицы.
4. Проведите несколько вычислительных экспериментов. Изучите зависимость времени выполнения алгоритма от объема исходных данных и от количества узлов, процессоров и ядер.

**2. Умножение матрицы на вектор при разделении данных по столбцам.** Другой подход к параллельному умножению матрицы на вектор основан на разделении исходной матрицы на непрерывные наборы (вертикальные полосы) столбцов. Вектор  $b$  при таком подходе разделен на блоки. Вертикальные полосы исходной матрицы и блоки вектора распределены между процессорами вычислительной системы.

Параллельный алгоритм умножения матрицы на вектор начинается с того, что каждый процессор  $i$  выполняет умножение своей вертикальной полосы матрицы  $A$  на блок элементов вектора  $b$ , в итоге на каждом процессоре получается блок вектора промежуточных результатов  $c'(i)$ . Далее для получения элементов результирующего вектора  $c$  процессоры должны обменяться своими промежуточными данными между собой.

**3. Умножение матрицы на вектор при блочном разделении данных.** В Паралаб реализован еще один параллельный алгоритм умножения матрицы на вектор, который основан на ином способе разделения данных – на разбиении матрицы на прямоугольные фрагменты (*блоки*). При таком способе разделения данных исходная матрица  $A$  представляется в виде набора прямоугольных блоков. Вектор  $b$  также должен быть разделен на блоки. Блоки матрицы и блоки вектора распределены между процессорами вычислительной системы. Логическая (виртуальная) топология вычислительной системы в данном случае имеет вид прямоугольной двумерной решетки. Размеры процессорной решетки соответствуют количеству прямоугольных блоков, на которые разбита матрица  $A$ . На процессоре  $p_{ij}$ , расположенному на пересечении  $i$ -ой строки и  $j$ -ого столбца процессорной решетки располагается блок  $A_{ij}$  матрицы  $A$  и блок  $b_j$  вектора  $b$ .

После перемножения блоков матрицы  $A$  и вектора  $b$  каждый процессор  $p_{ij}$  будет содержать вектор частичных результатов  $c'(i,j)$ . Поэлементное суммирование векторов частичных результатов для каждой горизонтальной строки процессорной решетки позволяет получить результирующий вектор  $c$ .

### **Задания и упражнения**

1. Запустите систему Параллаб. Установите количество процессоров, равное четырем.
2. Выполните три последовательных эксперимента с использованием трех различных алгоритмов умножения матрицы на вектор - алгоритмов, основанных на горизонтальном, вертикальном и блочном разбиении матрицы. Сравните временные характеристики алгоритмов, которые отображаются в правой нижней части окна.
3. Измените объем исходных данных (выполните команду **Параметры задачи** пункта меню **Задача**). Снова проведите эксперименты. Сравните временные характеристики алгоритмов.
4. Измените количество процессоров, установите количество узлов, равное 16 (выполните команду **Количество процессоров** пункта меню **Система**). Проведите вычислительные эксперименты и сравните временные характеристики. Сделайте выводы о сравнительной эффективности методов матрично-векторного умножения.

#### **13.4.2. Матричное умножение**

*Задача умножения матрицы на матрицу* определяется соотношениями:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, 1 \leq i, j \leq n$$

(для простоты изложения материала будем предполагать, что перемножаемые матрицы  $A$  и  $B$  являются квадратными и имеют порядок  $n \times n$ ). Как следует из приведенных соотношений, вычислительная сложность задачи является достаточно высокой (оценка количества выполняемых операций имеет порядок  $n^3$ ).

Основу возможности параллельных вычислений для матричного умножения составляет независимость расчетов для получения элементов  $c_{ij}$  результирующей матрицы  $C$ . Тем самым, все элементы матрицы  $C$  могут быть вычислены параллельно при наличии  $n^2$  процессоров, при этом на каждом процессоре будет располагаться по одной строке матрицы  $A$  и одному столбцу матрицы  $B$ . При меньшем количестве процессоров подобный подход приводит к *ленточной схеме* разбиения данных, когда на процессорах располагаются по несколько строк и столбцов (полос) исходных матриц.

Другой широко используемый подход для построения параллельных способов выполнения матричного умножения состоит в использовании *блочного представления* матриц, при котором исходные матрицы  $A$ ,  $B$  и результирующая матрица  $C$  рассматриваются в виде наборов блоков (как правило, квадратного вида некоторого размера  $m \times m$ ). Тогда операцию матричного умножения матриц  $A$  и  $B$  в блочном виде можно представить следующим образом:

$$\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1k} \\ \vdots & \vdots & & \vdots \\ A_{k1} & A_{k2} & \dots & A_{kk} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1k} \\ \vdots & \vdots & & \vdots \\ B_{k1} & B_{k2} & \dots & B_{kk} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1k} \\ \vdots & \vdots & & \vdots \\ C_{k1} & C_{k2} & \dots & C_{kk} \end{pmatrix},$$

где каждый блок  $C_{ij}$  матрицы  $C$  определяется в соответствии с выражением:

$$C_{ij} = \sum_{l=1}^k A_{il} B_{lj}.$$

Полученные блоки  $C_{ij}$  также являются независимыми и, как результат, возможный подход для параллельного выполнения вычислений может состоять в выполнении расчетов, связанных с получением отдельных блоков  $C_{ij}$ , на разных процессорах. Применение подобного подхода позволяет получить многие *эффективные параллельные методы умножения блочно-представленных матриц*.

В системе Паралаб реализованы *параллельный алгоритм умножения матриц при ленточной схеме разделения данных* и два метода (*алгоритмы Фокса и Кэннона*) для блочно-представленных матриц. Более полная информация об алгоритмах умножения матриц, реализованных в системе Паралаб, содержится в Главе 7.

**1. Ленточный алгоритм.** При ленточной схеме разделения данных исходные матрицы разбиваются на горизонтальные (для матрицы  $A$ ) и вертикальные (для матрицы  $B$ ) полосы. Получаемые полосы распределяются по процессорам, при этом на каждом из имеющегося набора процессоров располагается только по одной полосе матриц  $A$  и  $B$ . Перемножение полос (а выполнение процессорами этой операции может быть выполнено параллельно) приводит к получению части блоков результирующей матрицы  $C$ . Для вычисления оставшихся блоков матрицы  $C$  сочетания полос матриц  $A$  и  $B$  на процессорах должны быть изменены. В наиболее простом виде это может быть обеспечено, например, при кольцевой топологии вычислительной сети (при числе процессоров, равном количеству полос) – в этом случае необходимое для матричного умножения изменение положения данных может быть обеспечено циклическим сдвигом полос матрицы  $B$  по кольцу. После многократного выполнения описанных действий (количество необходимых повторений является равным числу процессоров) на каждом процессоре получается набор блоков, образующий горизонтальную полосу матрицы  $C$ .

Рассмотренная схема вычислений позволяет определить параллельный алгоритм матричного умножения при ленточной схеме разделения данных как итерационную процедуру, на каждом шаге которой происходит параллельное выполнение операции перемножения полос и последующего циклического сдвига полос одной из матриц по кольцу. Подробное описание ленточного алгоритма приводится в Главе 7.

### **Задания и упражнения**

1. Создайте в системе Паралаб новое окно вычислительного эксперимента. Для этого окна выберите задачу матричного умножения (щелкните левой кнопкой мыши на строке **Матричное умножение** пункта меню **Задача**).
2. Откройте диалоговое окно выбора топологии и выберете топологию кольцо.
3. Откройте диалоговое окно выбора метода и убедитесь в том, что выбран метод ленточного умножения матриц.
4. Проведите несколько вычислительных экспериментов. Изучите зависимость времени выполнения алгоритма от объема исходных данных и от количества узлов, процессоров и ядер.

**2. Блочные алгоритмы Фокса и Кэннона.** При блочном представлении данных параллельная вычислительная схема матричного умножения в наиболее простом виде может быть представлена, если топология вычислительной сети имеет вид прямоугольной решетки (если реальная топология сети имеет иной вид, представление сети в виде решетки можно обеспечить на логическом уровне). Основные положения параллельных методов для блочно представленных матриц состоят в следующем:

- Каждый из процессоров решетки отвечает за вычисление одного блока матрицы  $C$ ,
- В ходе вычислений на каждом из процессоров располагается по одному блоку исходных матриц  $A$  и  $B$ ,

- При выполнении итераций алгоритмов блоки матрицы  $A$  последовательно сдвигаются вдоль строк процессорной решетки, а блоки матрицы  $B$  – вдоль столбцов решетки,
- В результате вычислений на каждом из процессоров вычисляется блок матрицы  $C$ , при этом общее количество итераций алгоритма равно  $\sqrt{p}$  (где  $p$  – число процессоров),

В Главе 7 приводится полное описание параллельных методов Фокса и Кэннона для умножения блочно-представленных матриц.

### **Задания и упражнения**

1. В активном окне вычислительного эксперимента системы ПароЛаб установите топологию **Решетка**. Выберите число узлов, равное девяти. Сделайте текущей задачей этого окна задачу матричного умножения.
2. Выберите метод Фокса умножения матриц и проведите вычислительный эксперимент.
3. Выберите алгоритм Кэннона матричного умножения и выполните вычислительный эксперимент. Пронаблюдайте различные маршруты передачи данных при выполнении алгоритмов. Сравните временные характеристики алгоритмов.
4. Измените число узлов в топологии **Решетка** на шестнадцать. Последовательно выполните вычислительные эксперименты с использованием метода Фокса и метода Кэннона. Сравните временные характеристики этих экспериментов.

#### **13.4.3. Решение систем линейных уравнений**

*Системы линейных уравнений* возникают при решении ряда прикладных задач, описываемых системами нелинейных (трансцендентных), дифференциальных или интегральных уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

*Линейное уравнение* с  $n$  неизвестными  $x_0, x_1, \dots, x_{n-1}$  может быть определено при помощи выражения

$$a_0x_0 + a_1x_1 + \dots + a_{n-1}x_{n-1} = b$$

где величины  $a_0, a_1, \dots, a_{n-1}$  и  $b$  представляют собой постоянные значения.

Множество  $n$  линейных уравнений

$$\begin{array}{llll} a_{0,0}x_0 & +a_{0,1}x_1 & +\dots+a_{0,n-1}x_{n-1} & =b_0 \\ a_{1,0}x_0 & +a_{1,1}x_1 & +\dots+a_{1,n-1}x_{n-1} & =b_1 \\ \dots & & & \\ a_{n-1,0}x_0 & +a_{n-1,1}x_1 & +\dots+a_{n-1,n-1}x_{n-1} & =b_{n-1} \end{array}$$

называется *системой линейных уравнений* или *линейной системой*. В более кратком (*матричном*) виде система может быть представлена как

$$Ax = b,$$

где  $A=(a_{ij})$  есть вещественная матрица размера  $n \times n$ , а вектора  $b$  и  $x$  состоят из  $n$  элементов.

Под *задачей решения системы линейных уравнений* для заданных матрицы  $A$  и вектора  $b$  обычно понимается нахождение значения вектора неизвестных  $x$ , при котором выполняются все уравнения системы.

В системе ПароЛаб реализованы два алгоритма решения систем линейных уравнений: широко известный *метод Гаусса* (прямой метод решения систем линейных уравнений, находит точное решение системы с невырожденной матрицей за конечное число шагов) и

*метод сопряженных градиентов* – один из широкого класса итерационных методов решения систем линейных уравнений с матрицей специального вида. Более полная информация об алгоритмах решения систем линейных уравнений, реализованных в системе Паралаб, содержится в Главе 8.

**1. Алгоритм Гаусса.** *Метод Гаусса* включает последовательное выполнение двух этапов. На первом этапе – *прямой ход метода Гаусса* – исходная система линейный уравнений при помощи последовательного исключения неизвестных (при помощи эквивалентных преобразований) приводится к верхнему треугольному виду. На *обратном ходе метода Гаусса* (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной  $x_{n-1}$ , после этого из предпоследнего уравнения становится возможным определение переменной  $x_{n-2}$  и т.д.

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В этом случае матрицу  $A$  можно разделить на горизонтальные полосы, а вектор правых частей  $b$  – на блоки. Полосы матрицы и блоки вектора правых частей распределяются между процессорами вычислительной системы.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить  $(n-1)$  итерацию по исключению неизвестных для преобразования матрицы коэффициентов  $A$  к верхнему треугольному виду.

Выполнение итерации  $i$ ,  $0 \leq i < n-1$ , прямого хода метода Гаусса включает ряд последовательных действий. Процессор, на котором расположена строка  $i$  матрицы, должен разослать ее и соответствующий элемент вектора  $b$  всем процессорам, которые содержат строки с номерами  $k$ ,  $k > i$ . Получив ведущую строку, процессоры выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной  $x_i$ .

При выполнении **обратного хода** метода Гаусса процессоры выполняют необходимые вычисления для нахождения значения неизвестных. Как только какой-либо процессор определяет значение своей переменной  $x_i$ , это значение должно быть разослано всем процессорам, которые содержат строки с номерами  $k$ ,  $k < i$ . Далее процессоры подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора  $b$ .

### **Задания и упражнения**

1. В активном окне вычислительного эксперимента системы Паралаб установите топологию **Полный Граф**. Выберите число узлов, равное десяти. Сделайте текущей задачей этого окна задачу решения системы линейных уравнений.

2. Выберите метод Гаусса решения задачи и выполните вычислительный эксперимент. Пронаблюдайте маршруты передачи данных при выполнении алгоритма.

**2. Метод сопряженных градиентов.** *Метод сопряженных градиентов* – один из широкого класса итерационных методов решения систем линейных уравнений, который может быть применен для решения симметричной положительно определенной системы линейных уравнений большой размерности. При выполнении итераций метода сопряженных градиентов к искомому точному решению  $x^*$  системы  $Ax=b$  строится последовательность приближенных решений  $x^0, x^1, \dots, x^k, \dots$ . При этом процесс вычислений организуется таким способом, что каждое очередное приближение дает оценку точного решения со все уменьшающейся погрешностью, и при продолжении расчетов оценка точного решения может быть получена с любой требуемой точностью.

*Итерация метода сопряженных градиентов* состоит в вычислении очередного приближения к точному решению в соответствии с правилом:

$$x^k = x^{k-1} + s^k d^k,$$

в соответствии с которым новое значение приближения  $x^k$  вычисляется с учетом приближения, построенного на предыдущем шаге  $x^{k-1}$ , скалярного шага  $s^k$  и вектора направления  $d^k$ .

Перед выполнением первой итерации вектора  $x^0$  и  $d^0$  полагаются равными нулю, а для вектора  $g^0$  устанавливается значение равное  $-b$ . Далее каждая итерация для вычисления очередного значения приближения  $x^k$  включает выполнение четырех шагов:

**Шаг 1:** Вычисление градиента:

$$g^k = A \cdot x^{k-1} - b;$$

**Шаг 2:** Вычисление вектора направления:

$$d^k = -g^k + \frac{((g^k)^T, g^k)}{((g^{k-1})^T, g^{k-1})} d^{k-1},$$

где  $(g^T, g)$  есть скалярное произведение векторов;

**Шаг 3:** Вычисление величины смещения по выбранному направлению:

$$s^k = \frac{(d^k, g^k)}{(d^k)^T \cdot A \cdot d^k};$$

**Шаг 4:** Вычисление нового приближения:

$$x^k = x^{k-1} + s^k d^k.$$

В целом, сложность алгоритма имеет порядок  $O(n^3)$ .

При разработке параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений в первую очередь следует учесть, что выполнение итераций метода осуществляется последовательно и, тем самым, наиболее целесообразный подход состоит в распараллеливании вычислений, реализуемых в ходе выполнения отдельных итераций. Основные вычисления, выполняемые в соответствии с методом, состоят в умножении матрицы  $A$  на вектора  $x$  и  $d$ , и, как результат, при организации параллельных вычислений могут быть использованы параллельные алгоритмы умножения матрицы на вектор.

### Задания и упражнения

1. В активном окне вычислительного эксперимента системы ПароЛаб установите топологию **Полный Граф**. Выберите число процессоров, равное десяти. Сделайте текущей задачей этого окна задачу решения системы линейных уравнений.

2. Выберите метод сопряженных градиентов решения задачи и выполните вычислительный эксперимент. Пронаблюдайте маршруты передачи данных при выполнении алгоритма. Сравните эффективность алгоритма с эффективностью метода Гаусса.

#### 13.4.4. Сортировка данных

*Сортировка* является одной из типовых проблем обработки данных, и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, \dots, a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a'_1, a'_2, \dots, a'_n) : a'_1 \leq a'_2 \leq \dots \leq a'_n\}.$$

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T_1 \sim n^2.$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$$T_1 \sim n \log_2 n.$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из  $n$  значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких ( $p$ ,  $p > 1$ ) процессоров. Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе сортировки данные пересыпаются между процессорами и сравниваются между собой. Результирующий (упорядоченный) набор, как правило, также разделен между процессорами, при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

В системе Паралаб в качестве методов упорядочения данных представлены *пузырьковая сортировка*, *сортировка Шелла*, *быстрая сортировка*. Исходные (последовательные) варианты этих методов изложены во многих изданиях (см., например, [71]), способы их параллельного выполнения излагаются в главе 9.

**1. Пузырьковая сортировка.** Алгоритм *пузырьковой сортировки* в прямом виде достаточно сложен для распараллеливания: сравнение пар соседних элементов происходит строго последовательно. Параллельный вариант алгоритма основывается на *методе чет – нечетной перестановки*, для которого на нечетной итерации выполнения сравниваются элементы пар

$$(a_1, a_2), (a_3, a_4), \dots, (a_{n-1}, a_n).$$

Если пара не упорядочена, то ее элементы переставляются. На четной итерации упорядочиваются пары

$$(a_2, a_3), (a_4, a_5), \dots, (a_{n-2}, a_{n-1}).$$

После  $n$ -кратного повторения подобных итераций массив оказывается отсортированным. Более подробная информация о параллельном варианте алгоритма приводится в главе 9.

### Задания и упражнения

1. Запустите систему Паралаб и создайте новый эксперимент. Выберите пункт меню **Задача** и убедитесь, что в активном окне текущей задачей является задача сортировки. Откройте диалоговое окно выбора метода и посмотрите, какие алгоритмы сортировки могут быть выполнены на текущей топологии. Так как при создании эксперимента по умолчанию текущей топологией становится решетка, то единственный возможный алгоритм – алгоритм сортировки пузырьком. Закройте диалоговое окно и вернитесь в основное меню системы Паралаб.

2. Выполните несколько экспериментов, изменяя размер исходных данных. Для выполнения эксперимента выполните команду **В активном окне** пункта меню

**Выполнить.** Проанализируйте временные характеристики экспериментов, которые отображаются в правой нижней части окна.

3. Проведите несколько вычислительных экспериментов, изменяя количество узлов, процессоров и ядер вычислительной системы. Проанализируйте полученные временные характеристики.

**2. Сортировка Шелла.** *Параллельный алгоритм сортировки Шелла* может быть получен как обобщение метода параллельной пузырьковой сортировки. Основное различие состоит в том, что на первых итерациях алгоритма Шелла происходит сравнение пар элементов, которые в исходном наборе данных находятся далеко друг от друга (для упорядочивания таких пар в пузырьковой сортировке может понадобиться достаточно большое количество итераций). Подробное описание параллельного обобщения алгоритма сортировки Шелла можно найти в разделе 9.

### **Задания и упражнения**

1. Запустите систему Паралаб. Выберите топологию кольцо и установите количество узлов, равное восьми. Проведите эксперимент по выполнению алгоритма пузырьковой сортировки.

2. Установите в окне вычислительного эксперимента топологию гиперкуб и число узлов, равное восьми.

3. Откройте диалоговое окно выбора метода и посмотрите, какие алгоритмы сортировки могут быть выполнены на этой топологии. Выберите метод сортировки Шелла. Закройте окно.

4. Проведите вычислительный эксперимент. Сравните количество итераций, выполненных при решении задачи при помощи метода Шелла, с количеством итераций алгоритма пузырьковой сортировки (количество итераций отображается внизу в строке состояния). Убедитесь в том, что при выполнении эксперимента с использованием алгоритма Шелла, для сортировки массива необходимо выполнить меньшее количество итераций.

5. Проведите эксперимент с использованием метода Шелла несколько раз. Убедитесь, что количество итераций не является постоянной величиной и зависит от исходного массива.

**3. Быстрая сортировка.** *Алгоритм быстрой сортировки* основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности. При параллельном обобщении алгоритма обеспечивается отношение упорядоченности между элементами сортируемого набора, расположенными на процессорах, соседних в структуре гиперкуба. В главе 9 Вы можете найти подробное описание алгоритма быстрой сортировки.

### **Задания и упражнения**

1. Запустите систему Паралаб. В появившемся окне вычислительного эксперимента установите топологию гиперкуб и количество узлов, равное восьми.

2. Выполните три последовательных эксперимента с использованием трех различных алгоритмов сортировки: сортировки пузырьком, сортировки Шелла и быстрой сортировки. Сравните временные характеристики алгоритмов, которые отображаются в правой нижней части окна. Убедитесь в том, что у быстрой сортировки наименьшее время выполнения алгоритма и время передачи данных.

3. Измените объем исходных данных (выполните команду **Параметры задачи** пункта меню **Задача**). Снова проведите эксперименты. Сравните временные характеристики алгоритмов.

4. Измените количество процессоров (выполните команду **Количество процессоров** пункта меню **Система**). Проведите вычислительные эксперименты и сравните временные характеристики.

#### 13.4.5. Обработка графов

*Математические модели в виде графов* широко используются при моделировании самых разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор типовых алгоритмов обработки графов.

В системе Паралаб реализованы параллельные алгоритмы решения двух типовых задач на графах: *алгоритм Прима для поиска минимального охватывающего дерева* и *алгоритм Дейкстры для поиска кратчайших путей*. Более полная информация об алгоритмах обработки графов, реализованных в системе Паралаб, содержится в Главе 10.

#### Правила использования системы Паралаб

**1. Переход в режим редактирования графа.** При выборе задачи **Обработка графов** в системе Паралаб предусмотрена возможность создания и редактирования графов, запоминания графов в файле и чтения графа из файла. Для того чтобы перейти в режим редактирования графа, выполните команду **Редактор графов** пункта меню **Задача**. Заметим, что команда доступна только в том случае, когда текущей задачей является задача обработки графов.

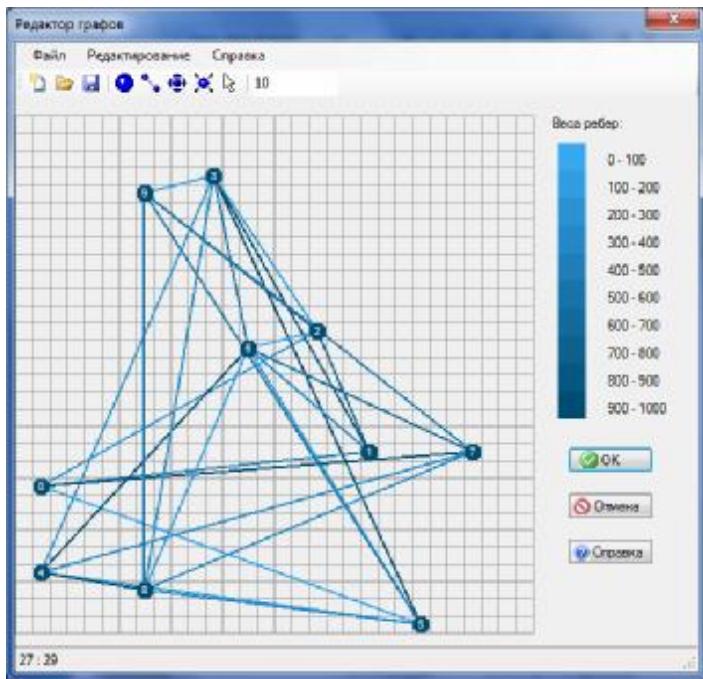


Рис. 13.10. Окно встроенного редактора графов

После выполнения команды **Редактор графа** на экране дисплея появляется новое окно (рис. 13.10), в рабочей области которого отображается граф активного эксперимента. Если график в эксперименте не был загружен, то рабочая область окна пуста.

Вам предоставляется возможность создавать новые графы, редактировать уже существующие, сохранять новые графы в файл и загружать график в активный эксперимент.

**2. Создание нового графа.** Для создания нового пустого графа выберите пункт меню **Файл** и выполните команду **Создать** (или щелкните левой кнопкой мыши на иконке  панели инструментов). Если граф, который отображается в рабочей области, был изменен, то Вам будет предложено сохранить измененный график в файл.

**3. Открытие существующего графа.** Для загрузки графа из файла выберите пункт меню **Файл** и выполните команду **Открыть** (или щелкните левой кнопкой мыши на иконке  панели инструментов). В появившемся диалоговом окне выберите имя файла (файлы графов Паралаб имеют расширение .plg) и нажмите кнопку **Открыть**.

**4. Сохранение графа.** Для сохранения графа в файл выберите пункт меню **Файл** и выполните команду **Сохранить** (или щелкните левой кнопкой мыши на иконке  панели инструментов). В появившемся диалоговом окне введите имя нового файла или выберите какой-либо из существующих файлов для того, чтобы сохранить график в этом файле. Нажмите кнопку **Сохранить**.

**5. Редактирование графа.** С графиком, расположенным в рабочей области окна, можно производить следующие операции:

- Для того, чтобы добавить к графу одну или несколько новых вершин, выберите пункт меню **Редактирование** и выполните команду **Добавить вершину** (или щелкните левой кнопкой мыши на иконке  панели инструментов). При этом вид курсора изменится, над указателем появится символическое изображение вершины. Рабочая область окна представляет собой сетку. Щелкнув левой кнопкой мыши в различных клетках сетки, Вы можете добавлять в график новые вершины. Если Вы щелкнули в клетке, где уже есть вершина, то добавления вершины не произойдет.

- Для того, чтобы соединить две вершины графа ребром, выберите пункт меню **Редактирование** и выполните команду **Добавить/удалить ребро** (или щелкните левой кнопкой мыши на иконке  панели инструментов). После этого курсор изменит форму, под указателем появится изображение двух вершин, соединенных ребром. Выделите одну из вершин графа, щелкнув на ней левой кнопкой мыши. Ее цвет изменится на темно-красный. Выделите другую вершину графа – в результате между первой и второй указанными вершинами появится ребро. Вес ребра определяется случайным образом. Если между первой и второй вершинами до редактирования существовало ребро, то оно будет удалено.

- Для того, чтобы переместить вершину графа, выберите пункт меню **Редактирование** и выполните команду **Переместить вершину** (или щелкните левой кнопкой мыши на иконке  панели инструментов). После этого курсор изменит форму, примет вид, изображенный на кнопке панели инструментов. Выделите одну из вершин графа, щелкнув на ней левой кнопкой мыши. Цвет вершины изменится на темно-красный. Перемещайте курсор мыши по рабочей области окна – Вы увидите, что вершина перемещается вслед за курсором. Щелкните на любой пустой клетке рабочей области, и выделенная вершина переместится в эту точку.

- Для того, чтобы удалить вершину графа, выберите пункт меню **Редактирование** и выполните команду **Удалить вершину** (или щелкните левой кнопкой мыши на иконке  панели инструментов). После этого курсор изменит вид, под указателем появится пиктограмма перечеркнутой вершины. Щелкните левой кнопкой мыши на любой вершине графа, чтобы удалить ее.

Для выхода из любого из режимов (Добавление вершины, Удаление вершины, Перемещение вершины, Добавление ребра) щелкните левой кнопкой мыши на иконке  панели инструментов.

**6. Формирование графа при помощи случайного механизма.** Граф можно задавать при помощи случайного генератора. Для этого в редакторе, расположенном на панели

инструментов, укажите число вершин графа, далее выберите пункт меню **Редактирование** и выполните команду **Создать случайный граф**.

**7. Редактирование веса ребра графа.** Цвет ребер графа имеет разную интенсивность. Чем темнее цвет, тем больше вес ребра. Для того, чтобы приблизительно определить вес ребра, нужно сравнить его цвет со шкалой, расположенной справа. Для того, чтобы изменить вес ребра, щелкните на нем правой кнопкой мыши. Рядом с ребром появится ползунок. Перемещая его вправо, Вы увеличиваете вес ребра, перемещая влево – уменьшаете. Для закрепления изменений щелкните мышкой в любой точке рабочей области или нажмите любую клавишу.

**8. Выход из режима редактирования.** Для загрузки текущего графа в активный эксперимент, нажмите кнопку **OK**. Для выхода без сохранения изменений нажмите кнопку **Отмена**.

**1. Алгоритм Прима поиска минимального охватывающего дерева.** *Охватывающим деревом (или остовом) неориентированного графа  $G$  называется подграф  $T$  графа  $G$ , который является деревом и содержит все вершины из  $G$ . Определив вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, тогда под минимально охватывающим деревом (МОД)  $T$  будем понимать охватывающее дерево минимального веса.* Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием наименьшего количества соединительных линий связи.

Дадим краткое описание алгоритма решения поставленной задачи, известного под названием *метода Прима (Prim)*. Алгоритм начинает работу с произвольной вершины графа, выбираемого в качестве корня дерева, и в ходе последовательно выполняемых итераций расширяет конструируемое дерево до МОД. Распределение данных между процессорами вычислительной системы должно обеспечивать независимость перечисленных операций алгоритма Прима. В частности, это может быть обеспечено, если каждая вершина графа располагается на процессоре вместе со всей связанной с вершиной информацией.

С учетом такого разделения данных итерация параллельного варианта алгоритма Прима состоит в следующем:

- определяется вершина, имеющая наименьшее расстояние до построенного к этому моменту МОД (операции вычисления расстояния для вершин графа, не включенных в МОД, независимы и, следовательно, могут быть выполнены параллельно);
- найденная вершина включается в состав МОД.

**2. Алгоритм Дейкстры поиска кратчайших путей.** *Задача поиска кратчайших путей* на графике состоит в нахождении путей минимального веса от некоторой заданной вершины  $s$  до всех имеющихся вершин графа. Постановка подобной проблемы имеет важное практическое значение в различных приложениях, когда веса дуг означают время, стоимость, расстояние, затраты и т.п.

Возможный способ решения поставленной задачи, известный как алгоритм Дейкстры, практически совпадает с методом Прима. Различие состоит лишь в интерпретации и в правиле оценки расстояний. В алгоритме Дейкстры эти величины означают суммарный вес пути от начальной вершины до всех остальных вершин графа. Как результат, на каждой итерации алгоритма выбирается очередная вершина, расстояние от которой до корня дерева минимально, и происходит включение этой вершины в дерево кратчайших путей.

### **Задания и упражнения**

1. Запустите систему Паралаб. В активном окне вычислительного эксперимента установите топологию **Полный граф**. Текущей задачей этого окна сделайте задачу обработки графов.
2. Выполните команду **Редактор графов** пункта меню **Задача**. В появившемся редакторе графов сформируйте случайным образом граф с десятью вершинами.
3. Выполните вычислительный эксперимент по поиску минимального охватывающего дерева с помощью алгоритма Прима (выполните команду **Метод решения** пункта меню **Задача**, в появившемся диалоговом окне выберите **Алгоритм Прима ( поиск охватывающего дерева)**).
4. Проведите несколько экспериментов, изменяя количество узлов, процессоров и ядер. Изучите зависимость временных характеристик алгоритма Прима от количества процессоров.
5. Проведите аналогичную последовательность экспериментов для изучения временных характеристик метода Дейкстры.

#### **13.4.6. Решение дифференциальных уравнений**

Дифференциальные уравнения в частных производных представляют собой широко применяемый математический аппарат при разработке моделей в самых разных областях науки и техники. К сожалению, явное решение этих уравнений в аналитическом виде оказывается возможным только в частных простых случаях, и, как результат, возможность анализа математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения. Объем выполняемых при этом вычислений обычно является значительным и использование высокопроизводительных вычислительных систем является традиционным для данной области вычислительной математики.

Приведем краткую информацию по решению дифференциальных уравнений в системе Паралаб; полная информация может получена в Главе 11.

В системе Паралаб, в качестве учебного примера рассматривается *проблема численного решения задачи Дирихле для уравнения Пуассона*, определяемую как задачу нахождения функции  $u = u(x, y)$ , удовлетворяющей в области определения  $D$  уравнению

$$\begin{cases} \frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = f(x, y), & (x, y) \in D, \\ u(x, y) = g(x, y), & (x, y) \in D^0, \end{cases}$$

и принимающей значения  $g(x, y)$  на границе  $D^0$  области  $D$  ( $f$  и  $g$  являются функциями, задаваемыми при постановке задачи). Подобная модель может быть использована для описания установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и деформации упругих пластин и др.

Для простоты изложения материала в качестве области задания  $D$  функции  $u(x, y)$  используется единичный квадрат

$$D = \{(x, y) \in D : 0 \leq x, y \leq 1\}.$$

Одним из наиболее распространенных подходов численного решения дифференциальных уравнений является *метод конечных разностей* (*метод сеток*). Следуя этому подходу, область решения  $D$  представляется в виде дискретного (как правило, равномерного) набора (*сетки*) точек (узлов). Так, например, прямоугольная сетка в области  $D$  может быть задана в виде (рис. 13.11)

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih, y_i = jh, 0 \leq i, j \leq N+1, \\ h = 1/(N+1), \end{cases}$$

где величина  $N$  задает количество узлов по каждой из координат области  $D$ .

Обозначим оцениваемую при подобном дискретном представлении аппроксимацию функции  $u(x, y)$  в точках  $(x_i, y_j)$  через  $u_{ij}$ . Тогда, используя *пятиточечный шаблон* (см. рис. 13.11) для вычисления значений производных, мы можем представить уравнение Пуассона в *конечно-разностной форме*

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}.$$

Данное уравнение может быть разрешено относительно  $u_{ij}$

$$u_{ij} = 0.25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Разностное уравнение, записанное в подобной форме, позволяет определять значение  $u_{ij}$  по известным значениям функции  $u(x, y)$  в соседних узлах используемого шаблона. Данный результат служит основой для построения различных *итерационных схем* решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений  $u_{ij}$ , а затем эти значения последовательно уточняются в соответствии с приведенным соотношением.

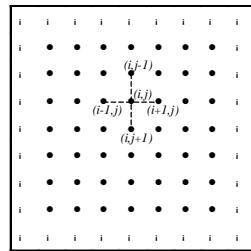


Рис. 13.11. Прямоугольная сетка в области  $D$  (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах - сверху вниз)

Для организации параллельных вычислений на системах с распределенной памятью необходимо выбрать способ разделения обрабатываемых данных между вычислительными серверами. Успешность такого разделения определяется достигнутой степенью локализации вычислений на серверах (в силу больших временных задержек при передаче сообщений интенсивность взаимодействия серверов должна быть минимальной).

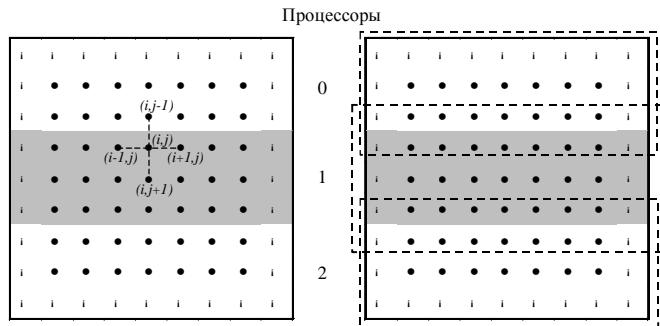


Рис. 13.12. Ленточное разделение области расчетов между процессорами (кружки представляют граничные узлы сетки)

В рассматриваемой учебной задаче по решению задачи Дирихле применяется ленточная схема разделения данных (см. рис. 13.12) вычислительной сетки.

При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы (не уменьшая общности, далее будем рассматривать только горизонтальные полосы). Число полос определяется количеством процессоров, размер полос обычно является одинаковым, узлы горизонтальных границ (первая и последняя строки) включаются в первую и последнюю полосы соответственно. Полосы для обработки распределяются между процессорами.

Основной момент при организации вычислений с подобным разделением данных состоит в том, что на процессор, выполняющий обработку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки (получаемые в результате расширенные полосы показаны на рис. 13.12 справа пунктирными рамками). Продублированные граничные строки полос используются только при проведении расчетов, пересчет же этих строк происходит в полосах своего исходного месторасположения. Тем самым дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток.

Для решения задачи Дирихле необходимо задать *граничные условия* и *начальные значения* узловых элементов сетки. Для решения уравнения необходимо так же задать *количество узлов в сетке*. Длительность вычислений определяется требуемой *точностью* и *максимально-возможным количеством выполняемых итераций*. Все перечисленные параметры в системе Паралаб можно задать в *параметрах задачи* (рис. 13.13).

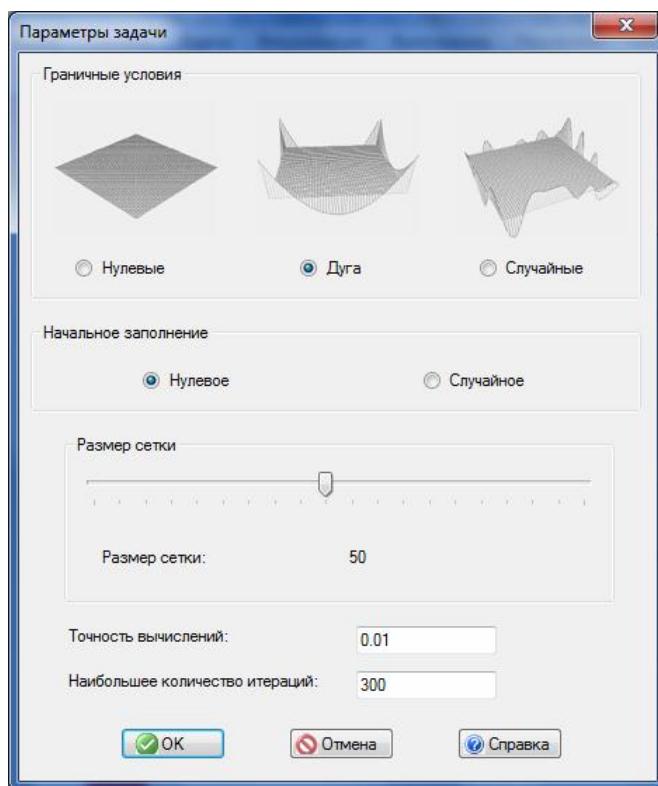


Рис. 13.13. Установка параметров задачи решения дифференциальных уравнений

### Задания и упражнения

1. Запустите систему Паралаб. В активном окне вычислительного эксперимента установите топологию **Решетка**. Текущей задачей этого окна установите задачу решения системы дифференциальных уравнений.

2. Задайте начальные условия для задачи решения дифференциального уравнения.
3. Выполните вычислительный эксперимент.
4. Проведите несколько экспериментов, изменяя количество узлов, процессоров и ядер. Изучите зависимость временных характеристик от количества процессоров и выбранной топологии узлов.

### 13.4.7. Решение задач многоэкстремальной оптимизации

Во всех областях своей целенаправленной деятельности перед человеком возникает проблема выбора наилучшего решения из множества всех возможных. Примерами могут служить экономика (управление экономическими объектами), техника (выбор оптимальной конструкции). К числу наиболее распространенных моделей рационального выбора относятся математические задачи оптимизации (максимизации или минимизации) некоторого функционала при ограничениях типа неравенств. При этом выполнение ограничений для некоторого вектора параметров, определяющего решение, интерпретируется как допустимость этого решения, т.е. как возможность его реализации при имеющихся ресурсах. Теорию и методы отыскания минимумов функций многих переменных при наличии дополнительных ограничений на эти переменные обычно рассматривают как отдельную дисциплину – математическое программирование.

Полное описание рассматриваемой темы приводится в Главе 12; здесь же дается краткий материал, достаточный для изучения методов многоэкстремальной оптимизации в системе Паралаб.

В общем виде задачу математического программирования можно сформулировать следующим образом. Пусть  $j(y)$ ,  $g_j(y) \leq 0$ ,  $1 \leq j \leq m$ , есть действительные функции, определенные на множестве  $D$   $N$ -мерного евклидова пространства  $R^N$ , и пусть точка  $y^*$  удовлетворяет условию

$$j(y^*) = \min\{j(y) : y \in D, g_j(y) \leq 0, 1 \leq j \leq m\}.$$

Точка из  $y^*$  обычно называется *глобально-оптимальной точкой* или *глобально-оптимальным решением*. При этом функцию  $j(y)$  называют *функцией цели*, или *целевой функцией*, а функции  $g_j(y) \leq 0$ ,  $1 \leq j \leq m$ , – *ограничениями задачи*.

Область  $D$  называют *областью поиска* и обычно описывают как некоторый гиперинтервал из  $N$ -мерного евклидова пространства

$$D = \{y \in R^N : a_i \leq y_i \leq b_i, 1 \leq i \leq n\},$$

где  $a, b \in R^N$  есть заданные векторы. Точки из области поиска, удовлетворяющие всем ограничениям, называются *допустимыми точками* или *допустимыми решениями*. Множество

$$Q = \{y : y \in D, g_j(y) \leq 0, 1 \leq j \leq m\}$$

всех таких точек называют *допустимой областью*.

Важный в прикладном отношении подкласс задач вида характеризуется тем, что все функционалы, входящие в определение задачи, заданы некоторыми (программно реализуемыми) алгоритмами вычисления значений  $j(y)$ ,  $g_j(y)$ ,  $1 \leq j \leq m$ , в точках области поиска  $D$ . При этом *решение задачи* сводится к построению оценки  $y_* \in Q$ , отвечающей некоторому понятию близости к точке  $y^*$  (например, чтобы  $\|y^* - y_*\| \leq e$  или  $|j(y^*) - j(y_*)| \leq e$ , где  $e > 0$  есть заданная точность) на основе некоторого числа  $k$  значений функционалов задачи, вычисленных в точках области  $D$ .

В задачах многоэкстремальной оптимизации возможность достоверной оценки глобального оптимума принципиально основана на наличии *априорной информации* о функции, позволяющей связать возможные значения минимизируемой функции с известными значениями в точках осуществленных поисковых итераций. Весьма часто такая априорная информация о задаче представляется в виде предположения, что целевая функция  $j$  (в дальнейшем обозначаемая также  $g_{m+1}$ ) и левые части ограничений  $g_j, 1 \leq j \leq m$ , удовлетворяют *условию Липшица* с соответствующими константами  $L_j, 1 \leq j \leq m+1$ , а именно

$$|g_j(y_1) - g_j(y_2)| \leq L_j \|y_1 - y_2\|, \quad 1 \leq j \leq m+1, \quad y_1, y_2 \in D.$$

В общем случае все эти функции могут быть *многоэкстремальными*.

Фундаментальные результаты в этом направлении были получены Нижегородской школой глобальной оптимизации (коллектив исследователей под руководством Р.Г. Стронгина, Нижегородский государственный университет). Один из наиболее важных результатов состоит в разработке информационно-статистического подхода к построению алгоритмов многоэкстремальной оптимизации [28,97]. Минимизируемая функция рассматривается как реализация некоторого случайного процесса, и решающие правила алгоритма конструируются таким образом, что очередная итерация проводится в точке глобального минимума математического ожидания значений функции. Более подробная информация о информационно-статистическом подходе приводится в Главе 12.

Для построения параллельного алгоритма используются множественные отображения (см. Главу 12). Использование множественных отображений позволяет решать исходную задачу путем параллельного решения индексным методом  $L+1$  задач на наборе отрезков  $[0, 1]$ . Каждая одномерная задача (или группа задач при недостаточном количестве процессоров) решается на отдельном процессоре. Результаты испытания в точке  $x^k$ , полученные конкретным процессором для решаемой им задачи, интерпретируются как результаты испытаний во всех остальных задачах (в соответствующих точках  $x^{k^0}, x^{k^1}, \dots, x^{k^L}$ ). При таком подходе испытание в точке  $x^k \in [0, 1]$ , осуществляющее в  $s$ -й задаче, состоит в последовательности действий:

1. Определить образ  $y^k = y^s(x^k)$  при соответствии  $y^s(x)$ .
2. Вычислить величину  $g_0(y^k)$ . Если  $g_0(y^k) \leq 0$ , то есть  $y^k \in D$ , то проинформировать остальные узлы о начале проведения испытания в точке  $y^k$  (*блокирование* точки  $y^k$ ).
3. Вычислить величины  $g_1(y^k), \dots, g_n(y^k)$ , где значения индекса  $n \leq m$  определяются условиями

$$g_j(y^k) \leq 0, \quad 1 \leq j < n, \quad g_n(y^k) > 0, \quad n \leq m.$$

Выявление первого нарушенного ограничения прерывает испытание в точке  $y^k$ . В случае, когда точка  $y^k$  допустима, т.е. когда  $y^s(x^k) \in Q_{m+1}$ , испытание включает вычисление значений всех функционалов задачи. При этом значение индекса принимается равным величине  $n=m+1$ , а тройка

$$y^s(x^k), \quad n=n(x^k), \quad z^k=g_n(y^s(x^k)),$$

является *результатом испытания* в точке  $x^k$ .

4. Если  $n(x^k) > 0$ , то есть  $y^k \in D$ , то определить прообразы  $x^{kl} \in [0, 1], 0 \leq l \leq L$ , точки  $y^k$ , и интерпретировать испытание, проведенное в точке  $y^k \in D$ , как проведение испытаний в  $L+1$  точке

$$x^{k^0}, x^{k^1}, \dots, x^{k^L},$$

с одинаковыми результатами

$$n(x^{k0})=n(x^{k1})=\dots=n(x^{kL})=n(x^k),$$

$$g_n(y^0(x^{k0}))=g_n(y^1(x^{k1}))=\dots=g_n(y^L(x^{kL}))=z^k.$$

Проинформировать остальные узлы о результатах испытания в точке  $y^k$ .

В случае если  $n(x^k)=0$ , то есть  $y^k \notin D$ , результат испытания относится только к  $s$ -й задаче.

Каждый узел имеет свою копию программных средств, реализующих вычисление функционалов задачи, и решающее правило алгоритма. Для организации взаимодействия на каждом узле создается  $L+1$  очередь, в которые процессоры помещают информацию о выполненных итерациях в виде троек: точка очередной итерации, индекс и значение, причем индекс заблокированной точки полагается равным  $-1$ , а значение функции в ней не определено.

Предлагаемая схема не содержит какого-либо единого управляющего узла, что увеличивает надежность выполняемых вычислений.

Для решения задачи оптимизации необходимо задать параметры, указанные в диалоговом окне Параметры задачи (см. рис. 13.14).

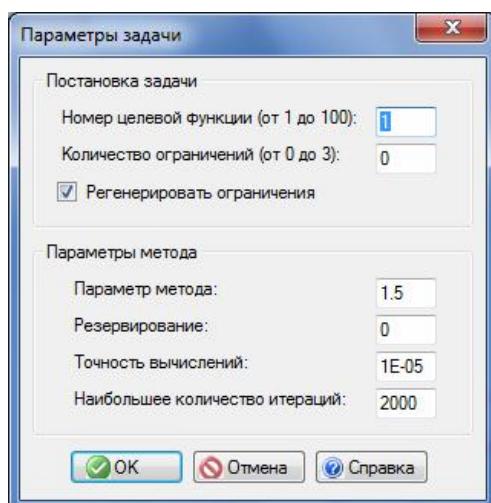


Рис. 13.14. Задание параметров задачи многоэкстремальной оптимизации

### **Задания и упражнения**

1. Запустите систему ПароЛаб. В активном окне вычислительного эксперимента установите топологию **Полный граф**. Текущей задачей этого окна сделайте задачу многоэкстремальной оптимизации.
2. Задайте начальные условия для задачи многоэкстремальной оптимизации.
3. Выполните вычислительный эксперимент по решению задачи многоэкстремальной оптимизации.
4. Проведите несколько экспериментов, изменяя количество узлов, процессоров и ядер. Изучите зависимость временных характеристик от количества процессоров, узлов и ядер.

### **13.5. Определение графических форм наблюдения за процессом параллельных вычислений**

Для наблюдения за процессом выполнения вычислительного эксперимента по параллельному решению сложных вычислительно трудоемких задач в рамках системы

ПараЛаб предусмотрены различные формы графического представления результатов выполняемых параллельных вычислений.

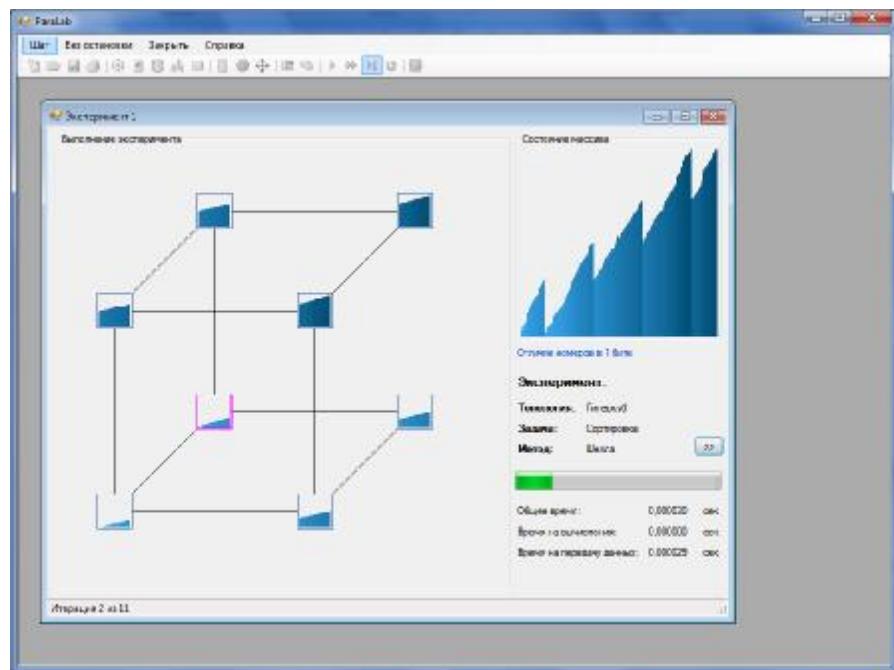


Рис. 13.15. Вид окна вычислительного эксперимента

Для представления сведений о ходе выполнения эксперимента в рабочей области системы ПараЛаб для каждого эксперимента выделяется прямоугольный участок экрана – *окно вычислительного эксперимента*. В левой части окна выделена область "Выполнение эксперимента", где изображаются процессоры многопроцессорной вычислительной системы, объединенные в ту или иную топологию, данные, расположенные на каждом процессоре, и взаимообмен данными между процессорами системы. В правой верхней части окна отображается область с информацией о текущем состоянии объекта, являющегося результатом выполняемого эксперимента. В зависимости от того, какой эксперимент выполняется, эта область носит название:

- "Результат умножения матрицы на вектор" при выполнении матрично-векторного умножения
- "Результат умножения матриц" при выполнении матричного умножения,
- "Состояние линейной системы" при решении системы линейных уравнений,
- "Текущее состояние массива" при выполнении алгоритма сортировки,
- "Результат обработки графа" при выполнении алгоритмов на графах,
- "Распределение тепла" при решении дифференциальных уравнений,
- "Целевая функция" при решении задачи многоэкстремальной оптимизации.

В средней части правой половины окна эксперимента приводятся сведения о выполняемой задаче.

В правом нижнем углу располагается ленточный индикатор выполнения эксперимента и его текущие временные характеристики.

Дополнительно в отдельном окне могут быть более подробно визуально представлены вычисления, которые производит один из имеющегося набора процессор (задание режима высветки этого окна и выбор наблюдаемого процессора осуществляется пользователем системы).

### 13.5.1. Область "Выполнение эксперимента"

В этой области окна изображены данные узлов многопроцессорной вычислительной системы, соединенные линиями коммутации в ту или иную топологию. Если дважды щелкнуть левой клавишей мыши на изображении данных ядра, то появится окно "Демонстрация работы ядра", где будет детально отображаться деятельность этого ядра.

Для каждого ядра схематически изображаются данные, которые находятся на нем в данный момент выполнения эксперимента. Если эксперимент состоит в изучении какого-либо параллельного алгоритма сортировки, то на каждом ядре изображается часть сортируемого массива. Каждый элемент массива изображается вертикальной линией. Высота и интенсивность цвета линии характеризуют величину элемента (чем выше и темнее линия, тем больше элемент). Если эксперимент заключается в изучении алгоритмов матричного умножения, то на каждом ядре изображен силуэт матрицы, на котором цветом выделены части исходных данных, располагаемых на ядре (предполагается, что исходные матрицы  $A$  и  $B$  являются квадратными порядка  $n \times n$ ). Синим цветом помечается часть матрицы  $A$  на ядре (блок или горизонтальная полоса), оранжевым – часть матрицы  $B$  (блок или вертикальная полоса). Если же эксперимент состоит в изучении алгоритмов обработки графов, то на каждом ядре изображается подграф, состоящий из вершин, расположенных на этом процессоре.

В процессе выполнения эксперимента в области "Выполнение эксперимента" также отображается обмен данными между процессорами многопроцессорной вычислительной системы. Это может происходить в двух режимах:

- режим "**Выделение линий связей**" - выделяется красным та линия коммутации, по которой происходит обмен,
- режим "**Пересылка пакетов**" - визуализация обмена при помощи движущегося от одного ядра к другому прямоугольника (конверта). Если изучаются параллельные алгоритмы матричного умножения, то на конверте изображается номер блока, который пересылается.

При выполнении алгоритмов на графах все итерации параллельного алгоритма однотипны и число их велико (равно числу вершин в графе). Для отображения всех итераций понадобится достаточно много времени. Поэтому в системе Паралаб реализована возможность отображать не все итерации, а лишь некоторые из них.

#### Правила использования системы Паралаб

**1. Изменение способа отображения пересылки данных.** Для задания способа отображения коммуникации процессоров выполните команду **Пересылка данных** пункта меню **Визуализация**. В появившемся списке выделите название желаемого способа отображения.

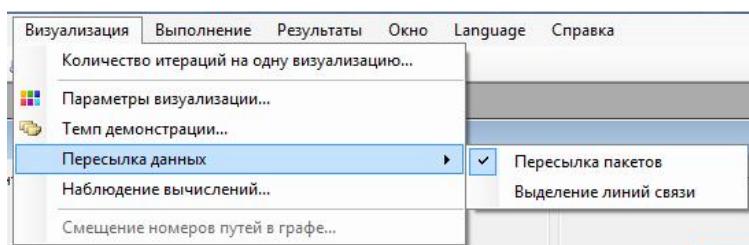


Рис. 13.16. Выбор способа отображения пересылки данных

**2. Выбор темпа демонстрации.** Для выбора темпа демонстрации необходимо выполнить команду **Темп Демонстрации** пункта меню **Визуализация**. В появившемся диалоговом окне (рис. 13.17) предоставляется возможность выбора величины задержки

между итерациями алгоритма и скорости движения пакетов (времени цветового выделения канала) при отображении коммуникации процессоров.

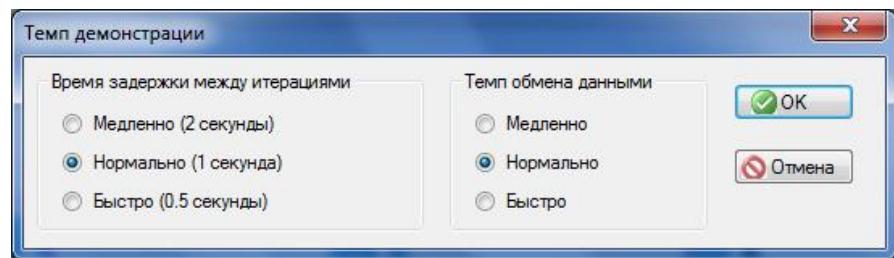


Рис. 13.17. Диалоговое окно выбора темпа демонстрации

Нажмите **OK** (Enter) для подтверждения выбора темпа демонстрации. Для возврата в основное меню системы ПароЛаб без сохранения изменений нажмите **Отмена** (Escape).

**3. Изменение шага визуализации.** Для того, чтобы в рабочей области системы ПароЛаб отображалась не каждая итерация, а лишь некоторые из них, выполните команду **Количество итераций на одну визуализацию** пункта меню **Визуализация**. В появившемся диалоговом окне установите при помощи ползунка желаемую частоту отображения итераций. Для выбора шага визуализации нажмите **OK** (Enter), для возврата в основное меню системы ПароЛаб нажмите **Отмена** (Escape)

**4. Настройка цветовой палитры.** Для изменения цветов, которые используются в системе ПароЛаб для визуализации процесса решения задач, выполните команду **Параметры визуализации** пункта меню **Визуализация**. В появившемся диалоговом окне (рис. 13.18) Для каждого алгоритма можно задать цвета отображения различных состояний данных, а также метод отображения результатов (двух- или трех- мерный). На приведенном рисунке представлен диалог для блочного перемножения матриц. Соответственно, матрицу можно отобразить только на двухмерной области и есть возможность задать цвет для вычисленных блоков и блоков, еще подлежащих вычислению.

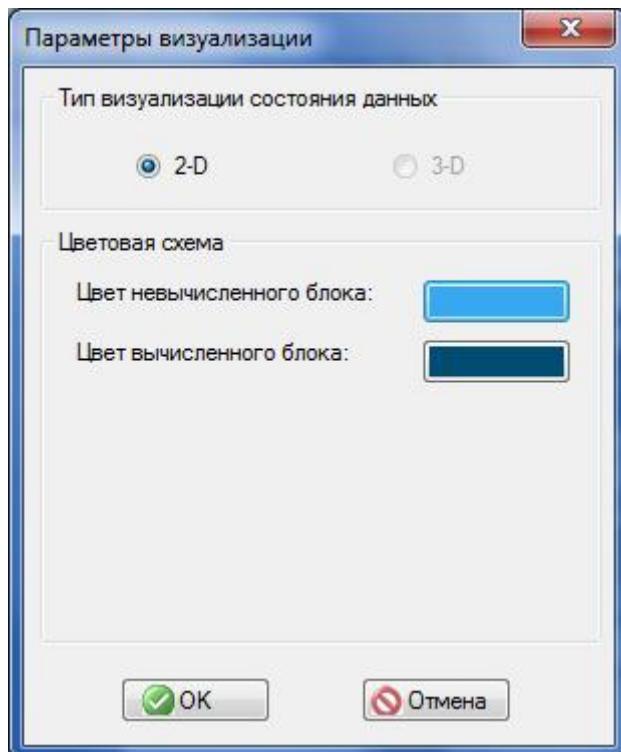


Рис. 13.18. Диалоговое окно настройки визуализации

Чтобы изменить цвета, щелкните левой клавишей мыши на цветной кнопке соответствующего параметра. В результате появится стандартное диалоговое окно выбора цвета операционной системы Windows. В этом окне выберите цвет и нажмите кнопку **OK**. Цвет будет изменен. Для того, чтобы изменить цвет выделения, щелкните левой клавишей мыши на отображающем этот цвет квадрате. При помощи стандартного диалогового окна задайте необходимый цвет.

Для изменения цветовой палитры нажмите кнопку **OK** (Enter) в диалоговом окне "Параметры визуализации". Для возврата в основной режим работы системы ПароЛаб нажмите **Отмена** (Escape).

### **13.5.2. Область "Текущее состояние массива"**

Эта область расположена в правой верхней части окна и отображает последовательность элементов сортируемого массива. Каждый элемент, как и в области "Выполнение эксперимента", отображается вертикальной линией. Высота и интенсивность цвета линии дают представление о величине элемента: чем выше и темнее линия, тем больше элемент.

Все параллельные алгоритмы используют идею разделения исходного массива между процессорами. Блоки, выстроенные один за другим в порядке возрастания номеров процессоров, на которых они располагаются, образуют результирующий массив. После выполнения сортировки блоки массива на каждом процессоре должны быть отсортированы и, кроме того, элементы, находящиеся на ядре с меньшим номером, не должны превосходить элементов, находящихся на ядре с большим номером.

Изначально массив – случайный набор элементов. После выполнения сортировки массив (при достаточно большом объеме исходных данных) изображается в виде прямоугольного треугольника с плавным перетеканием цвета из голубого в темно-синий.

### **13.5.3. Область "Результат умножения матрицы на вектор"**

Эта область находится в правой верхней части окна и отображает состояние вектора-результата в ядре выполнения параллельного алгоритма умножения матрицы на вектор.

При выполнении алгоритма, основанного на ленточном горизонтальном разбиении матрицы, каждое ядро вычисляет один блок результирующего вектора путем умножения полосы матрицы  $A$  на вектор-аргумент  $b$ . Вычисленный на активном ядре (подсвечен синим цветом) блок изображается темно-синим цветом. После выполнения коммуникации, на каждом ядре располагается весь результирующий вектор. Таким образом, все блоки вектора становятся темно-синими.

При выполнении алгоритма, основанного на ленточном вертикальном разбиении матрицы, каждое ядро вычисляет вектор частичных результатов путем умножения полосы матрицы на блок вектора-аргумента  $b$ ; все блоки результирующего вектора в области "Результат умножения матрицы на вектор" подсвечиваются светло-синим цветом. После выполнения коммуникационного шага на каждом ядре располагается блок результирующего вектора, блок активного процессора отображается в области "Результат умножения матрицы на вектор" темно-синим цветом.

При выполнении алгоритма, основанного на блочном разбиении матрицы, вектор  $b$  распределен между узлами, составляющими столбцы вычислительной системы. После умножения блока матрицы  $A$  на блок вектора  $b$  ядра процессоров вычисляет блок вектора частичных результатов – он подсвечивается светло-синим цветом. После обмена блоками в рамках одной строки узлов вычислительной системы, каждый узел этой строки содержит блок результирующего вектора, блок активного ядра отображается в области "Результат умножения матрицы на вектор" темно-синим цветом.

#### **13.5.4. Область "Результат умножения матриц"**

Эта область находится в правой верхней части окна и отображает состояние матрицы – результата в ядре выполнения параллельного алгоритма матричного умножения.

Матрица  $C$  представляется разбитой на квадратные блоки. Каждое ядро многопроцессорной вычислительной системы отвечает за вычисление одного (алгоритмы Фокса и Кэннона) или нескольких (ленточный алгоритм) блоков результирующей матрицы  $C$ .

При выполнении ленточного алгоритма умножения темно-синим цветом закрашиваются те блоки, которые уже вычислены к данному моменту.

Если же выполняется алгоритм Фокса или алгоритм Кэннона, то все блоки матрицы  $C$  вычисляются одновременно, ни один из блоков не может быть вычислен раньше, чем будут выполнены все итерации алгоритма. Поэтому в области "Результат умножения матриц" отображается динамика вычисления того блока результирующей матрицы, который расположен на активном ядре (это ядро в области "Выполнение эксперимента" выделен синим цветом). Вычисленные к этому моменту слагаемые написаны темно-синим цветом, вычисляемое на данной итерации – цветом выделения.



Рис. 13.19. Область "Результат умножения матриц" при выполнении алгоритма Фокса

#### **13.5.5. Область "Результат решения системы уравнений"**

Эта область расположена в правой верхней части окна вычислительного эксперимента и отображает текущее состояние матрицы линейной системы уравнений в ходе выполнения алгоритма Гаусса. Темно-синим цветом изображаются ненулевые элементы, а голубым – нулевые. После выполнения прямого хода алгоритма Гаусса ниже главной диагонали расположены только нулевые элементы. После выполнения обратного хода все ненулевые элементы расположены на главной диагонали.

#### **13.5.6. Область "Результат обработки графа"**

Эта область расположена в правой верхней части окна вычислительного эксперимента и отображает текущее состояние графа. Вершины графа имеют такое же взаимное расположение, как и в режиме редактирования графа. Дуги графа изображаются разными цветами: чем темнее цвет, тем больший вес имеет дуга.

В процессе выполнения алгоритмов на графах цветом выделения помечаются вершины и ребра, включенные к данному моменту в состав минимального охватывающего дерева (алгоритм Прима) или в дерево кратчайших путей (алгоритм Дейкстры).

#### **13.5.7. Область "Распределение тепла"**

Эта область расположена в правой верхней части окна вычислительного эксперимента и отображает текущее состояние узлов сетки разностной схемы. Каждый узел отображается точкой. Если точка маленькая, то это означает, что в узле на текущей итерации вычисления еще не выполнялись; если же узел выделен увеличенным маркером,

то вычисления в узле уже завершены. Цветами обозначена принадлежность тому или иному узлу сетки.

В процессе выполнения алгоритма динамически отображается последовательность обработки узлов.

### 13.5.8. Область "Целевая функция"

Эта область расположена в правой верхней части окна вычислительного эксперимента и отображает линии уровня и точки проведенных испытаний.

В процессе выполнения алгоритма поиска оптимального значения на графике линий уровня цветными точками отображаются точки испытаний, в которых выполнялось вычисления значения минимизируемой функции.

### 13.5.9. Выбор процессора

Для более детального наблюдения за процессом выполнения эксперимента в системе ПараЛаб предусмотрена возможность отображения вычислений одного из процессоров системы в отдельном окне.

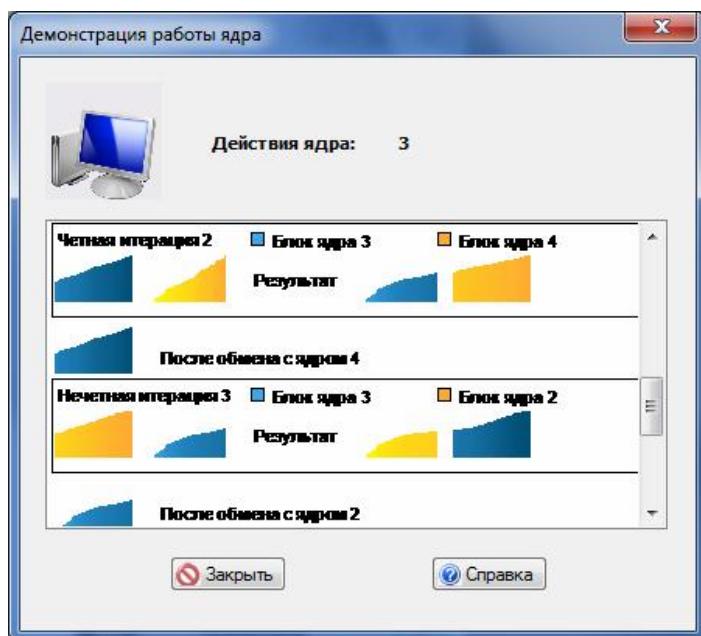


Рис. 13.20. Вид окна демонстрации работы процессора

Один из способов выбора процессора – выполнить команду **Наблюдение Вычислений** пункта меню **Визуализация**. В появившемся диалоговом окне при помощи бегунка укажите номер процессора и нажмите **OK (Enter)**. Для возврата в основное меню системы и отказа от выбора процессора нажмите **Отмена (Escape)**.

Второй способ выбора процессора – в рабочей области навести на него указатель мыши и щелкнуть дважды левой клавишей.

Далее в появившемся окне **"Демонстрация работы ядра"** будет детально изображаться ход вычислений.

## 13.6. Накопление и анализ результатов экспериментов

Выполнение численных экспериментов для изучения различных параллельных алгоритмов решения сложных вычислительных задач во многих случаях может потребовать проведения длительных вычислений. Для обоснования выдвигаемых предположений необходимо выполнить достаточно широкий набор экспериментов. Эти

эксперименты могут быть выполнены на различных многопроцессорных вычислительных системах, разными методами, для различных исходных данных. Для возможности сравнения результатов выполненных численных экспериментов система Паралаб обеспечивает ведение **Журнала экспериментов** и обеспечивает разнообразные способы представления данных журнала в виде форм, удобных для проведения анализа.

### 13.6.1. Общие результаты экспериментов

Накопление итогов экспериментов производится системой Паралаб автоматически. О каждом проведенном эксперименте хранится исчерпывающая информация: дата и время проведения, детальное описание вычислительной системы и решаемой задачи, время, потребовавшееся для выполнения эксперимента. Следует отметить, что повторение эксперимента с идентичными исходными установками приведет к повторному учету результатов.

При просмотре итогов предоставляется возможность восстановления эксперимента по сохраненной записи. Можно выполнять операции удаления записи и очистки списка итогов.

### 13.6.2. Просмотр итогов

Для отображения итогов экспериментов в системе Паралаб существует окно визуализации **Журнала экспериментов**. Данное окно содержит **Таблицу итогов** и **Лист графиков**.

Каждая строка **Таблицы итогов** (см. рис. 13.21) представляет один выполненный эксперимент. По умолчанию, в таблице итогов выделена первая строка и по ней в области **Лист графиков** построен график зависимости времени выполнения эксперимента от объема исходных данных на листе графиков. Для того чтобы изменить вид отображаемой зависимости, нужно выбрать соответствующие пункты в списках, расположенных в левом верхнем и нижнем правом углу листа графиков. Можно построить зависимости времени выполнения эксперимента и ускорения от объема исходных данных, количества узлов, количества процессоров в узле, количества ядер в процессоре, производительности процессоров и характеристик сети. Выделяя различные строки таблицы, Вы можете просматривать графики, соответствующие различным экспериментам.

Следует отметить, что при построении графиков для экспериментов, проведенных в режиме имитации, используются необходимые аналитические зависимости (см. Главы 6-10). Для экспериментов, проведенных на вычислительном кластере, используется набор полученных к данному моменту результатов реальных экспериментов.

При выделении нескольких строк в таблице результатов на листе графиков отображается несколько зависимостей. Цвет линии графика соответствует тому цвету, которым выделена левая ячейка строки, по которой построена эта зависимость.

При переходе к выполнению экспериментов, результаты которых не могут быть сопоставлены с итогами ранее проведенных вычислений, в системе Паралаб предусмотрена возможность очистить список итогов.

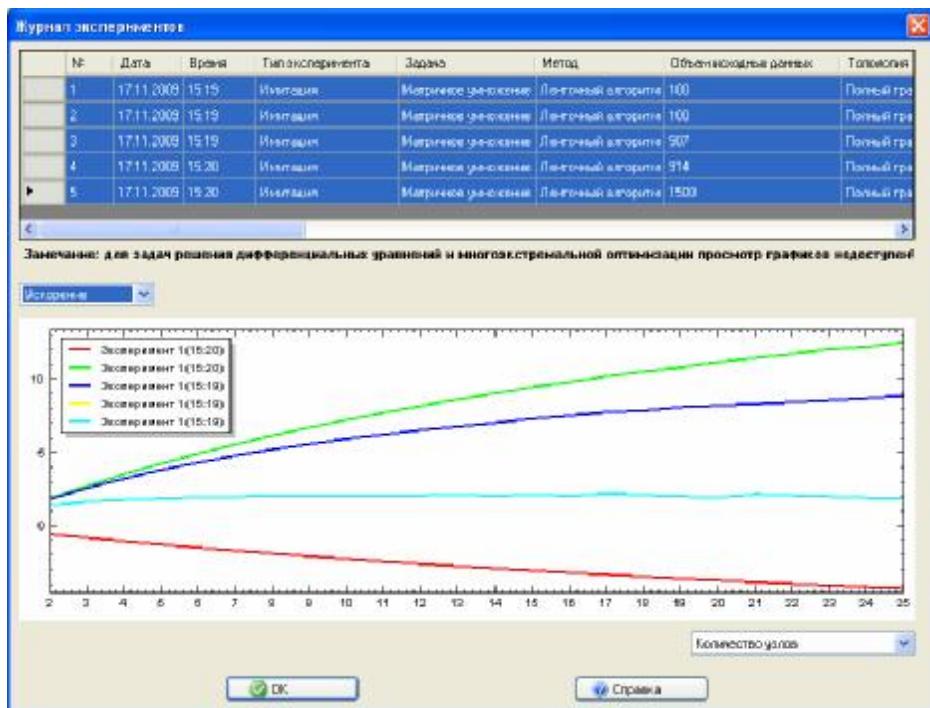


Рис. 13.21. Анализ результатов экспериментов

### *Правила использования системы *ПараЛаб**

**1. Общие результаты.** Для демонстрации накопленных результатов экспериментов следует выбрать пункт меню **Результаты**, выделить команду **Показать** и выполнить одну из двух команд: **Из активного окна** или **Из всех окон**. При выполнении первой команды будут отображены результаты, накопленные в активном окне вычислительного эксперимента. При выполнении второй команды – результаты из всех открытых окон экспериментов. Вид диалогового окна с результатами экспериментов представлен на рис. 13.21. Это окно содержит таблицу результатов и лист графиков.

**2. Выделение строки в таблице результатов.** Каждая строка таблицы представляет один выполненный эксперимент. Для выделения строки наведите указатель мыши на нужную строчку и нажмите левую кнопку мыши. Также можно воспользоваться курсорными стрелками вверх и вниз (выделенной строкой станет соответственно предыдущая или следующая строка). Если выделенная строчка одна, то на листе графиков отображается только зависимость, соответствующая выделенной строке.

**3. Выделение нескольких строк в таблице результатов.** Чтобы выделить несколько подряд идущих строк в таблице итогов, нажмите **Shift** и выделите мышью первую и последнюю строчку желаемого диапазона. Для выделения нескольких строк, не образующих непрерывную последовательность, нажмите **Ctrl** и выделяйте строки в произвольном порядке. Для того, чтобы выделить несколько строк при помощи курсорных клавиш, нажмите **Shift** и перемещайтесь по таблице при помощи клавиш вверх и вниз. При выделении нескольких строк в таблице результатов на листе графиков отображается несколько зависимостей. Цвет линии графика соответствует тому цвету, который сопоставлен в легенде проводимого эксперимента.

**4. Восстановление эксперимента по записи в таблице итогов.** Как уже отмечалось выше, запись в таблице итогов содержит исчерпывающую информацию о вычислительном эксперименте. Для восстановления эксперимента по записи необходимо выделить эту запись одним из перечисленных способов, щелкнуть правой кнопкой мыши в области списка итогов и выполнить команду **Восстановить эксперимент** появившегося контекстного меню. Эксперимент будет восстановлен в активном окне.

**5. Печать таблицы итогов.** Для печати списка итогов на печатающем устройстве щелкните правой кнопкой мыши в области таблицы и выполните команду **Печать** появившегося контекстного меню.

**6. Удаление записи.** Для удаления выделенной записи выполните команду **Удалить запись** контекстного меню списка итогов.

**7. Удаление результатов.** Для удаления накопленных результатов и перехода к построению новых оценок выполните команду **Очистить список** контекстного меню списка итогов.

**8. Изменение вида зависимости на листе графиков.** Для того, чтобы изменить вид зависимости, изображенной на листе графиков, выберите нужные значения в списках, расположенных слева вверху и справа внизу от листа графиков. Нижний правый список позволяет выбрать аргумент зависимости, а левый верхний – функцию.

**9. Копирование листа графиков в буфер обмена.** Для копирования графического изображения листа графиков в буфер обмена Windows выполните команду **Копировать в буфер обмена** контекстного меню.

**10. Печать листа графиков.** Для печати листа графиков на печатающем устройстве выполните команду **Печать** контекстного меню.

### **Задания и упражнения**

Выполните несколько экспериментов с одним и тем же методом умножения матриц, изменения объем исходных данных и количество процессоров. Используя окно итогов экспериментов, проанализируйте полученные результаты. Постройте одновременно несколько графиков на листе графиков и сравните их.

## **13.7. Выполнение вычислительных экспериментов**

В рамках системы Паралаб допускаются разные схемы организации вычислений при проведении экспериментов по изучению и исследованию параллельных алгоритмов решения сложных вычислительных задач. Решение задач может происходить в режиме последовательного исполнения или в режиме разделения времени с возможностью одновременного наблюдения итераций алгоритмов во всех окнах вычислительных экспериментов. Проведение серийных экспериментов, требующих длительных вычислений, может происходить в автоматическом режиме с возможностью запоминания результатов решения для организации последующего анализа полученных данных. Выполнение экспериментов может осуществляться и в пошаговом режиме.

### **13.7.1. Последовательное выполнение экспериментов**

В общем случае цель проведения вычислительных экспериментов состоит в оценке эффективности параллельного метода при решении сложных вычислительных задач в зависимости от параметров многопроцессорной вычислительной системы и от объема исходных данных. Выполнение таких экспериментов может сводиться к многократному повторению этапов постановки и решения задач. При решении задач в рамках системы Паралаб процесс может быть приостановлен в любой момент времени (например, для смены графических форм наблюдения за процессом решения) и продолжен далее до получения результата. Результаты решения вычислительных задач могут быть записаны в журнал экспериментов и представлены в виде, удобном для проведения анализа.

#### **Правила использования системы Паралаб**

**1. Проведение вычислительного эксперимента.** Для выполнения вычислительного эксперимента выберите пункт меню **Выполнение** и выполните команду **В активном**

**окне.** Решение задачи осуществляется без останова до получения результата. В ходе выполнения эксперимента основное меню системы заменяется на меню с командой **Остановить**; после завершения решения задачи основное меню системы восстанавливается.

**2. Приостановка решения.** Для приостановки процесса выполнения эксперимента следует выполнить в строке меню команду **Остановить** (команда доступна только до момента завершения решения).

**3. Продолжение решения.** Для продолжения ранее приостановленного процесса выполнения эксперимента следует выполнить команду **Продолжить** пункта меню **Выполнение** (команда может быть выполнена только в случае, если после приостановки процесса поиска не изменились постановка задачи и параметры вычислительной системы; при невозможности продолжения ранее приостановленного процесса выполнения эксперимента имя данной команды высвечивается серым цветом).

### **Задания и упражнения**

1. В активном окне вычислительного эксперимента установите топологию **Кольцо** и число процессоров, равное десяти. Сделайте текущей задачей задачу сортировки с использованием пузырькового алгоритма.
2. Выполните первые две итерации алгоритма и приостановите процесс вычислений.
3. Измените темп демонстрации и способ отображения пересылки данных.
4. Продолжите выполнение эксперимента до получения результата.

### **13.7.2. Выполнение экспериментов по шагам**

Для более детального анализа итераций параллельного алгоритма в системе Паралаб предусмотрена возможность пошагового выполнения вычислительных экспериментов. В данном режиме после выполнения каждой итерации происходит приостановка параллельного алгоритма. Это дает исследователю возможность подробнее изучить результаты проведенной итерации.

#### **Правила использования системы Паралаб**

**1. Пошаговый режим.** Для задания режима приостановки вычислительного эксперимента после выполнения каждой итерации следует выполнить команду **Пошаговый режим** пункта меню **Выполнение**. После выполнения этой команды основное меню системы Паралаб заменяется на меню пошагового выполнения эксперимента с командами:

- команда **Шаг** - выполнить очередную итерацию поиска,
- команда **Без Остановки** - продолжить выполнение эксперимента без остановки,
- команда **Закрыть** - приостановить выполнение эксперимента и вернуться к выполнению команд основного меню.

### **13.7.3. Выполнение нескольких экспериментов**

Последовательное выполнение экспериментов затрудняет сравнение результатов итераций параллельных алгоритмов. Для возможности более детального сравнения таких данных система Паралаб позволяет демонстрировать на экране дисплея одновременно результаты всех сравниваемых экспериментов. Для этого экран дисплея может разделяться на несколько прямоугольных областей (**окон экспериментов**), в каждой из которых могут высвечиваться результаты отдельно проводимого эксперимента. В любой момент пользователь системы Паралаб может создать новое окно для выполнения нового эксперимента. При этом итоги экспериментов и журнал экспериментов формируются

раздельно для каждого имеющегося окна. При визуализации окна экспериментов могут разделять экран (в этом случае содержимое всех окон является видимым) или могут перекрываться. Исследователь может выбрать любое окно активным для выполнения очередного эксперимента. Но вычисления могут быть выполнены и во всех окнах одновременно в режиме разделения времени, когда каждая новая итерация выполняется последовательно во всех имеющихся окнах. Используя этот режим, исследователь может наблюдать за динамикой выполнения нескольких экспериментов, результаты вычислений могут быть визуально различимы, и их сравнение может быть выполнено на простой наглядной основе.

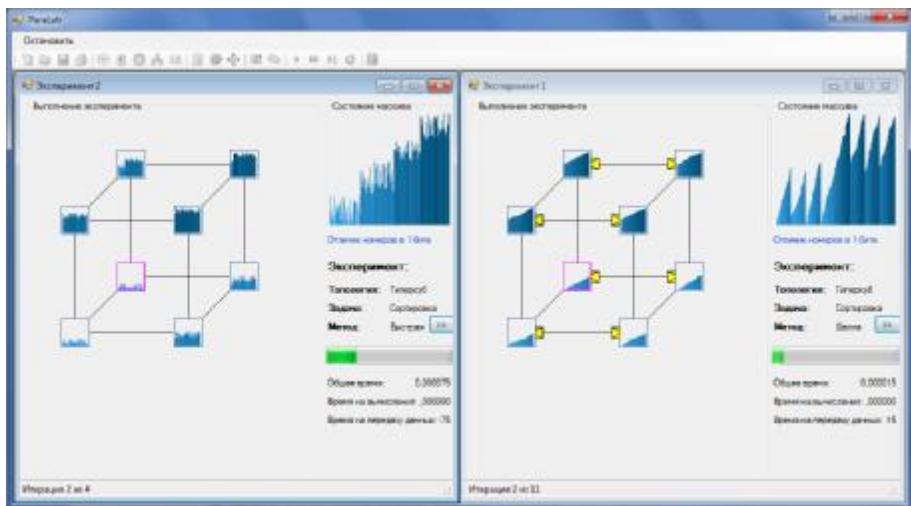


Рис. 13.22. Пример демонстрации нескольких окон экспериментов

Следует отметить, что итоги экспериментов, проведенных в разных окнах, могут высвечиваться совместно в одной и той же таблице итогов (см. п. 13.6.2). Это возможно и для данных из журналов экспериментов, соответствующих различным окнам.

#### *Правила использования системы ParaLab*

**1. Создание окна.** Для создания окна для проведения экспериментов следует выполнить команду **Создать новый** пункта меню **Архив**. Закрытие окна эксперимента производится принятymi в операционной системе Windows способами (например, путем нажатия кнопки закрытия окна в правом верхнем углу окна).

**2. Управление окнами.** Управление размерами окон экспериментов осуществляется принятими в системе Windows способами (максимизация, минимизация, изменение размеров при помощи мыши). Для одновременного показа всех имеющихся окон без перекрытия можно использовать команду **Показать все** пункта меню **Окно**; для выделения большей части экрана для активного окна (но при сохранении возможности быстрого доступа ко всем имеющимся окнам) следует применить команду **Расположить каскадом** пункта меню **Окно**.

**3. Проведение экспериментов во всех окнах.** Для выполнения вычислительных экспериментов во всех имеющихся окнах в режиме разделения времени (т.е. при переходе к выполнению следующей итерации только после завершения текущей во всех имеющихся окнах) следует применить команду **Во всех окнах** пункта меню **Выполнение**. Управление процессом вычислений осуществляется так же, как и при использовании единственного окна (приостановка выполнения алгоритмов по команде **Остановить**, продолжение вычислений по команде **Продолжить** пункта меню **Выполнение**).

**4. Сравнение итогов экспериментов.** Для того, чтобы свести в одну таблицу итогов результаты, полученные во всех окнах экспериментов, выполните последовательность команд **Результаты**→**Показать**→**Из всех окон**.

## **Задания и упражнения**

1. Откройте второе окно вычислительного эксперимента, установите режим показа окон без перекрытия.
2. В первом окне выберите метод пузырьковой сортировки. Во втором окне установите топологию **Гиперкуб** и выберите метод сортировки Шелла. Выполните копирование вычислительной системы в первое окно.
3. В обоих окнах установите режим автозаписи результатов в журнал экспериментов.
4. Выполните вычислительные эксперименты одновременно в обоих окнах; отрегулируйте скорость демонстрации установкой подходящего темпа показа.
5. Получите сводную таблицу итогов. Сравните временные характеристики алгоритмов пузырьковой сортировки и сортировки Шелла.

### **13.7.4. Выполнение серии экспериментов**

ПараЛаб обеспечивает возможность автоматического (без участия пользователя) выполнения серий экспериментов, требующих проведения длительных вычислений. При задании этого режима работы системы пользователь должен выбрать окно, в котором будут выполняться эксперименты, установить количество экспериментов и выбрать тот параметр, который будет изменяться от эксперимента к эксперименту (объем исходных данных или количество процессоров). Результаты экспериментов могут быть запомнены в журнале экспериментов, а в последующем проанализированы.

#### **Правила использования системы ПараЛаб**

**1. Выполнить серию.** Переход в режим выполнения последовательности экспериментов осуществляется при помощи команды **Серия экспериментов** пункта меню **Выполнение**. При выполнении команды может быть задано число экспериментов в серии, а также выбран тип серии: исследуется ли зависимость времени и ускорения решения поставленной задачи от объема исходных данных, от количества используемых процессоров, от количества процессоров в узле или от количества ядер на процессоре.

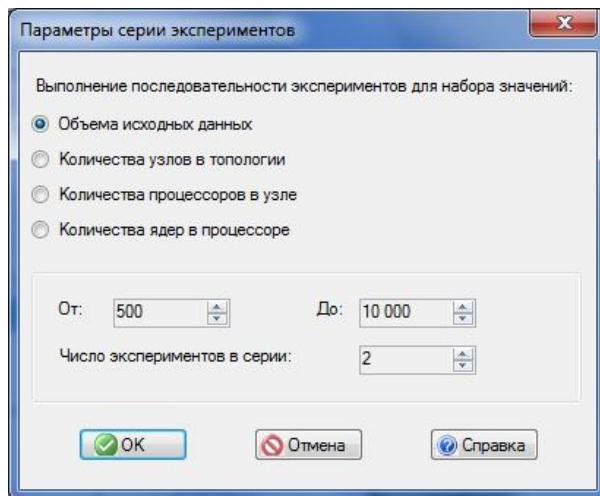


Рис. 13.23. Диалоговое окно задания параметров серии

При выполнении серии экспериментов основное меню системы ПараЛаб заменяется на меню управления данным режимом вычислений, команды которого позволяют:

- команда **Запуск** – выполнить последовательность экспериментов,
- команда **Закрыть** – приостановить выполнение данного режима и вернуться к выполнению команд основного меню,

- команда **Справка** – получение дополнительной справочной информации.

При решении серии поставленных задач (после выполнения команды **OK**) выполнение эксперимента может быть приостановлено в любой момент времени при помощи команды **Остановить**.

### 13.7.5. Выполнение реальных вычислительных экспериментов

Помимо выполнения экспериментов в режиме имитации, в системе Паралаб предусмотрена возможность проведения реальных экспериментов в режиме удаленного доступа к вычислительному кластеру. При выборе этого режима выполнения эксперимента необходимо поставить задачу и выбрать нужное количество процессоров для ее решения. После выполнения имитационных и реальных экспериментов пользователь Паралаб может сравнить результаты и оценить точность используемых в системе теоретических моделей времени выполнения параллельных алгоритмов. Результаты реальных экспериментов автоматически заносятся в таблицу итогов, кроме того, они могут быть запомнены в журнале экспериментов.

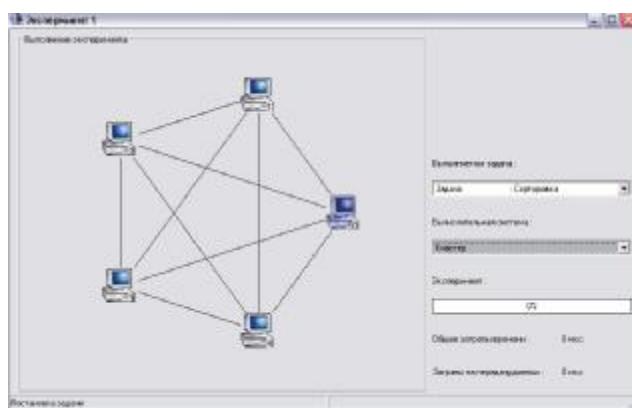


Рис. 13.24. Окно для выполнения реального вычислительного эксперимента

При выполнении реального эксперимента ход вычислений и обмен данными между процессорами не отображаются. В списке параметров вычислительной системы присутствует только строка, указывающая на то, что выполняется эксперимент в режиме удаленного доступа к кластеру, и указывается число процессоров. Режим пошагового выполнения эксперимента недоступен.

#### *Правила использования системы Паралаб*

**1. Переход в режим реального выполнения эксперимента.** Для перехода в режим выполнения реальных вычислительных экспериментов в режиме удаленного доступа к вычислительному кластеру выберите пункт меню **Система** и выделите мышью команду **Кластер**. Подтверждением того, что данный режим активен, является значок  слева от надписи. После выбора этого режима топология вычислительной системы автоматически заменяется на топологию **Полный граф**, так как последняя соответствует топологии кластера. Постановка задачи осуществляется так же, как и при выполнении экспериментов в режиме имитации.

**2. Задание количества вычислительных узлов (процессоров).** Для выбора числа процессоров выполните команду **Количество процессоров** пункта меню **Система**. В появившемся диалоговом окне при помощи бегунка задайте нужное число процессоров. Нажмите **OK** (Enter) для подтверждения выбора или **Отмена** (Escape) для возврата в основное меню системы без изменений.

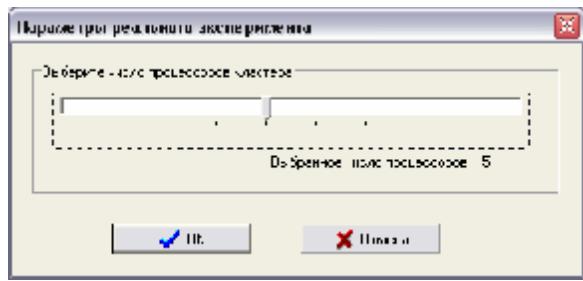


Рис. 13.25. Окно выбора количества вычислительных узлов

**3. Проведение реального эксперимента.** Для проведения реального вычислительного эксперимента выполните команду **В активном окне** пункта меню **Выполнение**.

### 13.8. Использование результатов экспериментов: запоминание, печать и перенос в другие программы

#### 13.8.1. Запоминание результатов

В любой момент результаты выполненных в активном окне вычислительных экспериментов могут быть сохранены в архиве системы Паралаб. Данные, сохраняемые для окна проведения эксперимента, включают:

- параметры активной вычислительной системы (топология, количество процессоров, производительность процессора, время начальной подготовки данных, пропускная способность сети, метод передачи данных),
- постановку задачи (тип задачи, размер исходных данных, метод решения).

Данные, сохраненные в архиве системы, в любой момент могут быть восстановлены из архива и, тем самым, пользователь может продолжать выполнение своих экспериментов в течение нескольких сеансов работы с системой Паралаб.

Кроме того, в рамках системы Паралаб исследователю предоставляется возможность сохранения в архиве и чтения из архива сформированных графов специального вида (см. п. 13.4.5).

#### *Правила использования системы Паралаб*

**1. Запись данных.** Для сохранения результатов выполненных экспериментов следует выполнить команду **Сохранить** пункта меню **Архив**. При выполнении записи в диалоговом окне следует задать имя файла, в котором будут сохранены данные. Расширение имени файла может не указываться. Файлы с параметрами вычислительных экспериментов имеют расширение **.prl**.

**2. Чтение данных.** Для чтения параметров экспериментов, записанных ранее в архив системы Паралаб, следует выбрать пункт меню **Архив** и указать команду **Загрузить**. После выполнения этой команды в активное окно будут загружены параметры вычислительного эксперимента, сохраненные в выбранном файле.

#### *Задания и упражнения*

Выполните вычислительные эксперименты, план проведения которых состоит в следующем:

1. Выполните какой-либо эксперимент и сохраните параметры выполненного эксперимента в архиве системы.
2. Завершите выполнение системы.

3. Выполните повторный запуск системы и загрузите запомненные параметры эксперимента из архива.

### **13.8.2. Печать результатов экспериментов**

При выполнении экспериментов в системе Паралаб получаемые результаты могут быть напечатаны в виде разнообразных графических и табличных форм. Пользователь системы может напечатать:

- таблицы итогов, сохраненных в журнале экспериментов,
- графические формы, являющиеся точными копиями содержимого окон проведения экспериментов,
- графические формы листа графиков из формы представления итогов экспериментов,
- графические формы, представляющие окно редактора графов.

Для печати результатов экспериментов могут быть использованы также стандартные возможности печати системы Windows при помощи копирования содержимого экрана.

#### ***Правила использования системы Паралаб***

**1. Печать таблицы результатов экспериментов.** Для печати таблицы следует открыть окно представления итогов экспериментов (выполнить последовательность команд: **Результаты**→**Показать**→**Из активного окна** или **Результаты**→**Показать**→**Из всех окон**), вызвать контекстное меню, связанное с таблицей итогов (щелкнуть правой кнопкой мыши в области таблицы), и выполнить команду **Печать**.

**2. Печать графической формы листа графиков.** Для печати листа графиков следует открыть окно представления итогов экспериментов, вызвать контекстное меню листа графиков и выполнить команду **Печать**.

**3. Печать окон экспериментов.** Для печати содержимого активного окна эксперимента следует выполнить команду **Печать** пункта меню **Архив**.

**4. Печать окна редактора графов.** Для печати содержимого окна редактора графов выполните команду **Печать** пункта меню **Файл** этого окна.

#### ***Задания и упражнения***

Выполните эксперименты и напечатайте графические формы окон экспериментов и таблицы итогов экспериментов.

### **13.8.3. Копирование результатов в другие программы**

При выполнении вычислительных экспериментов в системе Паралаб получаемые результаты могут быть скопированы в буфер обмена системы Windows в графическом формате и могут, тем самым, быть перемещены в любые другие программы системы Windows для последующего анализа и обработки. В буфер обмена могут быть скопированы:

- Графики зависимостей времени и ускорения от других параметров вычислительной системы.

Графическое представление результатов экспериментов может быть далее использовано в графических редакторах типа Paint, Photoshop или Corel Draw.

**Приложение: Таблица выполнимости методов на различных топологиях в системе Паралаб**

Метод	Линейка	Кольцо	Звезда	Решетка	Гиперкуб	Полный граф
<b>Умножение матрицы на вектор</b>						
Горизонтальное разбиение						
Вертикальное разбиение						
Блочное разбиение						
<b>Матричное умножение</b>						
Ленточный алгоритм						
Алгоритм Фокса						
Алгоритм Кэннона						
<b>Решение систем линейных уравнений</b>						
Алгоритм Гаусса						
Метод сопряженных градиентов						
<b>Сортировка</b>						
Пузырьковая						
Модифицированная пузырьковая						
Шелла						
Быстрая						
<b>Обработка графов</b>						
Алгоритм Прима						
Алгоритм Флойда						
Алгоритм Дейкстры						
<b>Решение диффер. уравнений</b>						
Метод Гаусса-Зейделя						
<b>Многоэкстремальная оптимизация</b>						
Индексный						

## **Литература**

1. Антонов А.С. "Параллельное программирование с использованием технологии OpenMP: Учебное пособие".- М.: Изд-во МГУ, 2009
2. Бахвалов Н.С. Численные методы. – М.: Наука, 1973
3. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. – М.: Наука, 1987
4. Березин И.С., Жидков Н.П. Методы вычислений. - М.: Наука, 1966
5. Бэкон Д., Харрис Т. Операционные системы. – СПб: Питер, 2004.
6. Васильев Ф.П. Методы оптимизации. М.: Факториал Пресс, 2002.
7. Вильямс А. Системное программирование в Windows 2000 для профессионалов. - СПб.: БХВ-Петербург, 2001.
8. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
9. Воеводин Вл.В., Жуматий С.А. Вычислительное дело и кластерные системы. - М.: Изд-во МГУ, 2007. - 161 с.
10. Гергель В.П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
11. Гергель В.П., Стронгин Л.Г., Стронгин Р.Г. Метод окрестностей в задачах распознавания // Изд. АН СССР. Техническая кибернетика. 1987. №4. С. 14-22.
12. Гергель, В.П., Стронгин, Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. - Н.Новгород, ННГУ (2 изд., 2003), 2001.
13. Городецкий С.Ю., Гришагин В.А. Нелинейное программирование и многоэкстремальная оптимизация. – Н.Новгород: Изд-во ННГУ, 2007.
14. Карманов В.Г. Математическое программирование. М.: Физматлит, 2004.
15. Карпов В.Е., Коньков К.А. Введение в операционные системы. Курс лекций. 2-е изд. – М.: ИНТУИТ.РУ, 2005.
16. Касперски К. Техника оптимизации программ. – СПб.: БХВ-Петербург, 2003.
17. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.:МСНТОБ 1999
18. Корнеев В.В. Параллельное программирование в MPI. Москва-Ижевск: Институт компьютерных исследований,2003
19. Лацис А. Как построить и использовать суперкомпьютер. М.: Бестселлер, 2003.
20. Маркин Д.Л., Стронгин Р.Г. Метод решения многоэкстремальных задач с невыпуклыми ограничениями, использующий априорную информацию об оценках оптимума // Журнал вычислительной математики и математической физики 1987. Т. 27. №1. С. 52-62
21. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем – СПб.: БХВ-Петербург, 2002.
22. Рихтер Дж. Windows для профессионалов (Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows). 4-е изд. – М.: Русская Редакция; пер. с англ. – СПб.: Питер, 2001.
23. Самарский А.А., Гулин А.В. Численные методы. – М.:Наука, 1989
24. Сиднев А.А., Камаев А.М., Сысоев А.В. Об одном подходе к оптимизации приложений // Высокопроизводительные параллельные вычисления на кластерных системах. Труды конференции. Казань: Изд. КГТУ. 2008. С. 39-42
25. Солomon Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. СПб: Издательский дом Питер, М.: Русская редакция, 2001
26. Стивенс У. UNIX: Взаимодействие процессов. СПб: Издательский дом Питер, 2002
27. Столлингс В. Операционные системы. М.: Вильямс, 2001
28. Стронгин Р.Г. Численные методы в многоэкстремальных задачах. М.: Наука, 1978.

29. Стронгин Р.Г., Маркин Д.Л. Минимизация многоэкстремальных функций при невыпуклых ограничениях // Кибернетика. 1986. №4. С. 64-69.
30. Стронгин Р.Г., Маркина М.В. Многоэкстремальные лексикографические задачи выбора // Всесоюзная школа-семинар. Системное моделирование процессов интенсификации общественного производства. Май 1987, Горький. Тезисы докладов. Секции 1, 2, 3. Горький: Изд-во ГГУ, 1987. С. 169-170.
31. Таненбаум Э. Современные операционные системы. 2-е изд. – СПб.: Питер, 2002.
32. Таненбаум Э., Ван Стейн М. Распределенные системы. Принципы и парадигмы. СПб.: Издательский дом Питер, 2003
33. Таненбаум Э., Вудхалл А. Операционные системы. СПб.: Питер, 2007.
34. Тихонов А.Н., Самарский А.А. Уравнения математической физики. – М.: Наука, 1977
35. Хамахер К., Вранешич З., Заки С. Организация ЭВМ. – СПб:Питер,2003
36. Шоу А. Логическое проектирование операционных систем. М.: Мир, 1981
37. Akl S.G. Parallel Sorting Algorithms. – Orlando, FL: Academic Press, 1984
38. Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings, Vol. 30, 1967, pp. 483-485, Washington, D.C.: Thompson Books.
39. Andrews, G. R. Foundations of Multithreaded, Parallel, and Distributed Programming.. – Reading, MA: Addison-Wesley, 2000 (русский перевод Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом "Вильямс", 2003)
40. Barkalov K.A., Strongin R.G. A global optimization technique with an adaptive order of checking for constraints. Computational mathematics and mathematical physics, vol. 42, №9, 2002, pp. 1289-1300.
41. Barnard S. PMRSB: Parallel multilevel recursive spectral bisection // Proc. Supercomputing '95. 1995
42. Barnard, S. PMRSB: Parallel multilevel recursive spectral bisection. In Proc. Supercomputing 95, 1995.
43. Berger, M., Bokhari, S. Partitioning strategy for nonuniform problems on multiprocessors. - IEEE Transactions on Computers, C-36(5). P. 570-580, 1987.
44. Bertsekas D.P., Tsitsiklis J.N. Parallel and Distribution Computation. Numerical Methods. – Prentice Hall, Englewood Cliffs, New Jersey, 1989
45. Blackford L.S., Choi J., Cleary A., D'Azevedo E., Demmel J., Dhillon I., Dongarra J.J., Hammarling S., Henry G., Petitet A., Stanley D., Walker R.C., Whaley K. Scalapack Users'Guide (Software, Environments, Tools). Soc. For Industrial & Applied Math., 1997
46. Butenhof, D. R. (1007) Programming with POSIX Threads. Boston, MA: Addison-Wesley Professional.,1997
47. Buyya R. (Ed.) High Performance Cluster Computing Volume 1: Architectures and Systems. Volume 2: Programming and Applications. – Prentice Hall PTR, Prentice-Hall Inc., 1999
48. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Melon, R.. Parallel Programming in OpenMP. San-Francisco, CA: Morgan Kaufmann Publishers., 2000
49. Culler, D., Singh, J.P., Gupta, A. Parallel Computer Architecture: A Hardware/Software Approach. - Morgan Kaufmann. , 1998.
50. Dongarra J.J., Duff L.S., Sorensen D.C., Vorst H.A.V. Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc. For Industrial & Applied Math., 1999
51. Fletcher R. Practical Methods of Optimization. Vol. 1,2. John Wiley and Sons, 1980.
52. Floudas C.A., Pardalos P.M. (Eds). Encyclopedia of Optimization. Kluwer Academic Publishers, Dordrecht, 2001.
53. Floudas C.A., Pardalos P.M. State of the Art in Global Optimization. Kluwer Academic Publishers, Dordrecht, 1996.

54. Foster I. Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley, 1995.
55. Fox G.C. et al. Solving Problems on Concurrent Processors.- Prentice Hall, Englewood Cliffs, NJ,1988
56. Geist G.A., Beguelin A., Dongarra J., Jiang W., Manchek B., Sunderam V. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing. MIT Press, 1994.
57. George, A., Liu, J. Computer Solution of Large Sparse Positive Definite Systems. - Prentice-Hall, Englewood Cliffs NJ, 1981.
58. Gilbert, J., Miller, G., Teng, S. Geometric mesh partitioning: Implementation and experiments. - In Proceedings of International Parallel Processing Symposium, 1995.
59. Gilbert, J., Zmijewski, E. A parallel graph partitioning algorithm for a message-passing multiprocessor. International Journal of Parallel Programming. 1987, pp. 498-513.
60. Gill P.E., Murray W., Wright M.E. Practical Optimization. Academic Press, London, 1981.
61. Grama, A.Y., Gupta, A. and Kumar, V. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. IEEE Parallel and Distributed technology. 1 (3). 1993, pp. 12-21.
62. Group W, Lusk E, Skjellum A. Using MPI. Portable Parallel Programming with the Message-Passing Interface. –MIT Press,1994
63. Gustavson, J.L. Reevaluating Amdahl's law. Communications of the ACM. 31 (5). 1988, pp.532-533.
64. Hansen E.R. Global Optimization Using Interval Analysis. Marcel Dekker, New York, 1992.
65. Heath, M., Raghavan, P. A Cartesian parallel nested dissection algorithm. SIAM Journal of Matrix Analysis and Applications, 16(1):235-253, 1995.
66. Himmelblau D.M. Applied Nonlinear Programming. McGraw-Hill, New York, 1972.
67. Hockney R.W., Jesshope C.R. Parallel Computers 2. Architecture, Programming and Algorithms. – Adam Hillger, Bristol and Philadelphia, 1988 (русский перевод 1-го издания. Хокни Р., Джессхоуп К. Параллельные ЭВМ. Архитектура программирование и алгоритмы. М.Ж Ради и связь, 1986).
68. Kahaner, D., Moler, C., Nash, S. Numerical Methods and Software. – Prentice Hall, 1988 (русский перевод Каханер Д., Моулер Л., Нэш С. Численные методы и программное обеспечение. М.: Мир, 2001)
69. Karypis, G., Kumar, V. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of Parallel and Distributed Computing, 48(1), 1998.
70. Karypis, G., Kumar, V. Parallel multilevel k-way partitioning scheme for irregular graphs. Siam Review, 41(2): 278-300, 1999.
71. Knuth, D. E. (1997). The Art of Computer Programming. Volume 3: Sorting and Searching, second edition. – Reading, MA: Addison-Wesley. (русский перевод Кнут Д. (2000). Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. 2 издание - М.: Издательский дом "Вильямс")
72. Kumar V., Grama A., Gupta A., Karypis G. Introduction to Parallel Computing , Inc. 1994 (2th edition, 2003)
73. Miller, G., Teng, S., Thurston, W., Vavasis, S. Automatic mesh partitioning. - In A. George, John R. Gilbert, and J. Liu, editors, Sparse Matrix Computations: Graph Theory Issues and Algorithms. IMA Volumes in Mathematics and its applications. Springer-Verlag, 1993.
74. Mockus J. et al. Bayesian Heuristic Approach to Discrete and Global Optimization: Algorithms, Visualization, Software, and Applications. Kluwer Academic Publishers, Dordrecht, 1996.
75. Nour-Omid, B., Raefsky, A., Lyzenga, G. Solving finite element equations on concurrent computers. - In A. K. Noor, editor, American Soc. Mech. P. 291-307, 1986.

76. Ou, C., Ranka, S., Fox G. Fast and parallel mapping algorithms for irregular and adaptive problems. - Journal of Supercomputing, 10. P. 119-140, 1996.
77. Patra, A., Kim, D. Efficient mesh partitioning for adaptive hp finite element methods. – In International Conference on Domain Decomposition Methods, 1998.
78. Patterson, D.A., Hennessy J.L. Computer Architecture: A Quantitative Approach. 2d ed. - San Francisco: Morgan Kaufmann., 1996.
79. Peano G. Sur une courbe, qui remplit toute une aire plane. Mathematische Annalen 36, 1890, P. 157-160.
80. Pfister, G.P. In Search of Clusters. Prentice Hall PTR, Upper Saddle River, NJ 1995.
81. Pilkington, J., Baden, S. Partitioning with spacefilling curves. Technical Report CS94-349, Dept. of Computer Science and Engineering, Univ. of California, 1994.
82. Pinter J. Global Optimization in Action (Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications. Kluwer Academic Publishers, Dordrecht, 1996.
83. Piyavskij S.A. An algorithm for finding the absolute extremum of a function. USSR Journal of Computational Mathematics and Mathematical Physics №12, pp. 57-67, 1972.
84. Pothen, A. Graph partitioning algorithms with applications to scientific computing. - In D. Keyes, A. Sameh, and V. Venkatakrishnan, editors, Parallel Numerical Algorithms. Kluwer Academic Press, 1996.
85. Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004
86. Raghavan, P. Line and plane separators. - Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61901, 1993.
87. Raghavan, P. Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee, 1995.
88. RightMark Memory Analyzer – cm. <http://cpu.rightmark.org>
89. Roosta, S.H. Parallel Processing and Parallel Algorithms: Theory and Computation. Springer-Verlag, NY. 2000.
90. Schloegel K., Karypis G., Kumar V. Graph Partitioning for High Performance Scientific Simulations. 2000
91. Sergeyev Ya.D., Grishagin V.A. Parallel Asynchronous Global Search and the Nested Optimization Scheme. Journal of Computational Analysis and Applications. Vol. 3, №2, 2001, P. 123-145.
92. Sterling, T. Beowulf Cluster Computing with Linux. - Cambridge, MA: The MIT Press. , (Ed.) 2002
93. Sterling, T. Beowulf Cluster Computing with Windows. - Cambridge, MA: The MIT Press., (Ed.) 2001
94. Strongin R.G. Algorithms for multi-extremal mathematical programming problems employing the set of joint space-filling curves// J. of Global Optimization, 1992. №2. P. 357–378.
95. Strongin R.G. Numerical methods for multiextremal nonlinear programming problems with nonconvex constraints// Lecture Notes in Economics and Mathematical Systems, 1985. V. 255. P. 278–282.
96. Strongin R.G., Sergeyev Ya.D. Global multidimensional optimization on parallel computer // Parallel Computing, 1992. V. 18. № 11. P.1259-1273.
97. Strongin R.G., Sergeyev Ya.D. Global optimization with non-convex constraints. Sequential and parallel algorithms. Kluwer Academic Publishers, Dordrecht, 2000.
98. Walshaw, C., Cross, M. Parallel optimization algorithms for multilevel mesh partitioning. Technical Report 99/IM/44, University of Greenwich, London, UK, 1999.
99. Wilkinson B., Allen M. Parallel programming. – Prentice Hall, 1999
100. Xu, Z., Hwang, K. Scalable Parallel Computing Technology, Architecture, Programming. McGraw-Hill, Boston.1998.

101. Zilinskas A. Axiomatic characterization of a global optimization algorithm and investigation of its search strategy. Operations Research Letters №4, 1985, P. 35-39.
102. Zomaya, A.Y. (Ed.) Parallel and Distributed Computing Handbook. - McGraw-Hill, 1996.

### **Информационные ресурсы сети Интернет**

103. «Мини-кластер в одном корпусе: 96 процессоров и 192 Гб RAM» – [<http://www.ixbt.com/staticnews/04/12/67.html>].
104. Будущие процессоры и технологии Intel: прорыв на новый уровень – [<http://www.intel.com/cd/corporate/pressroom/emea/rus/archive/2008/386850.htm>].
105. Персональный суперкомпьютер «Т-Платформы» поставлен в ННГУ – [<http://www.software.unn.ac.ru/?doc=673>].
106. Портал «Top 500® Supercomputers Sites» – [<http://www.top500.org>].
107. Intel VTune Performance - [<http://software.intel.com/en-us/intel-vtune>].
108. Web-сайт компании ClearSpeed Technology – [<http://www.clearspeed.com>].
109. Web-сайт компании Rendercube – [<http://www.rendercube.com>].
110. Web-сайт, посвященный библиотеке LinPack – [<http://www.netlib.org/lipack>].
111. Web-страница системы «ASCI Red» – [<http://www.sandia.gov/ASCI/Red>].
112. Web-страница системы Roadrunner на сайте «Top 500® Supercomputers Sites» – [<http://www.top500.org/system/9485>].

### **Источники рисунков**

1. Рис.1.5. – <http://www.overclockers.ru/lab/22660.shtml>.
2. Рис.1.6. – [http://www.amd.com/us-en/Processors/ProductInformation/0\\_30\\_118\\_15331\\_15332%5E15334,00.html](http://www.amd.com/us-en/Processors/ProductInformation/0_30_118_15331_15332%5E15334,00.html).
3. Рис.1.7. – <http://www-01.ibm.com/support/docview.wss?uid=tss1prs3036&aid=1>.
4. Рис.1.9. – [http://ru.wikipedia.org/wiki/UltraSPARC\\_T1](http://ru.wikipedia.org/wiki/UltraSPARC_T1).
5. Рис.1.10. – «ClearSpeed™ Advance™ X620 Product Datasheet».
6. Рис.1.11. – System Specification. NVIDIA Tesla D870 Deskside GPU Computing System – [[http://www.nvidia.com/docs/IO/43395/D870-SystemSpec-SP-03718-001\\_v01.pdf](http://www.nvidia.com/docs/IO/43395/D870-SystemSpec-SP-03718-001_v01.pdf)].
7. Рис. 1.12. – «Мини-кластер в одном корпусе: 96 процессоров и 192 Гб RAM» – [<http://www.ixbt.com/staticnews/04/12/67.html>].
8. Рис. 1.13. – «Готовые решения компании «Т-Платформы» для прикладных задач» – [[http://www.t-platforms.ru/pdf/T-Platforms\\_Turn-key\\_Solutions.pdf](http://www.t-platforms.ru/pdf/T-Platforms_Turn-key_Solutions.pdf)].