Построение вычислительных систем

Доктор технических наук В.П. Гергель.

УЧЕБНАЯ РАБОЧАЯ ПРОГРАММА по общему курсу "Многопроцессорные системы и параллельное программирование"

http://www.hpcc.unn.ru/files/HTML_Version/literature.html

1. Принципы построения параллельных вычислительных систем

1.1. Пути достижения параллелизма

В общем плане под *параллельными вычислениями* понимаются процессы обработки данных, в которых одновременно могут выполняться нескольких машинных операций. Достижение параллелизма возможно только при выполнимости следующих требований к архитектурным принципам построения вычислительной системы:

- **независимость функционирования отдельных устройств ЭВМ** - данное требование относится в равной степени ко всем основным компонентам вычислительной системы - к устройствам ввода-вывода, к обрабатывающим процессорам и к устройствам памяти;

- **избыточность элементов вычислительной системы - организация избыточности может осуществляться в следующих основных формах:**

  - *использование специализированных устройств* таких, например, как отдельных процессоров для целочисленной и вещественной арифметики, устройств многоуровневой памяти (регистры, кэш);

  - *дублирование устройств ЭВМ* путем использования, например, нескольких однотипных обрабатывающих процессоров или нескольких устройств оперативной памяти.

Дополнительной формой обеспечения параллелизма может служить **конвейерная реализация** обрабатывающих устройств, при которой выполнение операций в устройствах представляется в виде исполнения последовательности составляющих операцию подкоманд; как результат, при вычислениях на таких устройствах могут находиться на разных стадиях обработки одновременно несколько различных элементов данных.

Возможные пути достижения параллелизма детально рассматриваются в [22, 29]; в этой же работе рассматривается история развития параллельных вычислений и приводятся примеры конкретных параллельных ЭВМ (см. также [9, 29, 31]).

При рассмотрении проблемы организации параллельных вычислений следует различать следующие возможные режимы выполнения независимых частей программы:

- *многозадачный режим (режим разделения времени)*, при котором для выполнения процессов используется единственный процессор; данный режим является псевдопараллельным, когда активным (исполняемым) может быть один единственный процесс, а все остальные процессы находятся в состоянии ожидания своей очереди на использование процессора; использование режима разделения времени может повысить эффективность организации вычислений (например, если один из процессов не может выполняться из-за ожидании вводимых данных, процессор может быть задействован для готового к исполнению процесса - см. [6, 13]), кроме того, в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и др.) и, как результат, этот режим может быть использован при начальной подготовке параллельных программ;

• *параллельное выполнение*, когда в один и тот же момент времени может выполняться несколько команд обработки данных; данный режим вычислений может быть обеспечен не только при наличии нескольких процессоров, но реализуем и при помощи конвейерных и векторных обрабатывающих устройств;

• *распределенные вычисления*; данный термин обычно используют для указания параллельной обработки данных, при которой используется несколько обрабатывающих устройств, достаточно удаленных друг от друга и в которых передача данных по линиям связи приводит к существенным временным задержкам; как результат, эффективная обработка данных при таком способе организации вычислений возможна только для параллельных алгоритмов с низкой интенсивностью потоков межпроцессорных передач данных; перечисленные условия является характерными, например, при организации вычислений в *многомашинных вычислительных комплексах*, образуемых объединением нескольких отдельных ЭВМ с помощью каналов связи локальных или глобальных информационных сетей.

В рамках данного пособия основное внимание будет уделяться второму типу организации параллелизма, реализуемому на многопроцессорных вычислительных системах.

1.2. Классификация вычислительных систем

Одним из наиболее распространенных способов классификации ЭВМ является *систематика Флинна* (Flynn), в рамках которой основное внимание при анализе архитектуры вычислительных систем уделяется способам взаимодействия последовательностей (*потоков*) выполняемых команд и обрабатываемых данных. В результате такого подхода различают следующие основные типы систем [9, 22, 29, 31]:

• **SISD** (Single Instruction, Single Data) - системы, в которых существует одиночный поток команд и одиночный поток данных; к данному типу систем можно отнести обычные последовательные ЭВМ;

• **SIMD** (Single Instruction, Multiple Data) - системы с одиночным потоком команд и множественным потоком данных; подобный класс систем составляют МВС, в которых в каждый момент времени может выполняться одна и та же команда для обработки нескольких информационных элементов;

• **MISD** (Multiple Instruction, Single Data) - системы, в которых существует множественный поток команд и одиночный поток данных; примеров конкретных ЭВМ, соответствующих данному типу вычислительных систем, не существует; введение подобного класса предпринимается для полноты системы классификации;

• **MIMD** (Multiple Instruction, Multiple Data) - системы с множественным потоком команд и множественным потоком данных; к подобному классу систем относится большинство параллельных многопроцессорных вычислительных систем.

Следует отметить, что хотя систематика Флинна широко используется при конкретизации типов компьютерных систем, такая классификация приводит к тому, что практически все виды параллельных систем (несмотря на их существенную разнородность) относятся к одной группе MIMD. Как результат, многими исследователями предпринимались неоднократные попытки детализации систематики Флинна. Так, например, для класса MIMD предложена практически общепризнанная структурная схема [29, 31], в которой дальнейшее разделение типов многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах (см. рис. 1.1). 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).

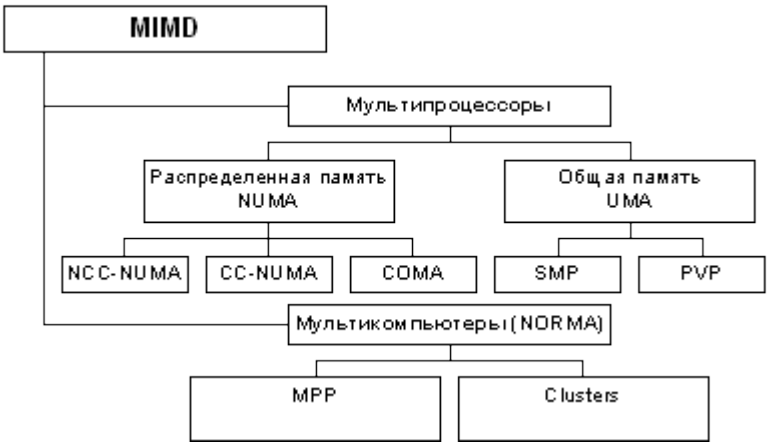

Рис. 1.1. Структура класса многопроцессорных вычислительных систем

Далее для *мультипроцессоров* учитывается способ построения общей памяти. Возможный подход - использование единой (централизованной) общей памяти. Такой подход обеспечивает *однородный доступ к памяти* (*uniform memory access or UMA*) и служит основой для построения *векторных суперкомпьютеров* (*parallel vector processor, PVP*) и симметричных *мультипроцессоров* (*symmetric multiprocessor or SMP*). Среди примеров первой группы суперкомпьютер Cray T90, ко второй группе относятся IBM eServer p690, Sun Fire E15K, HP Superdome, SGI Origin 300 и др.

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). 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 однозначность (*когерентность*) локальных кэш памяти разных процессоров (*cache-coherent NUMA or CC-NUMA*); среди систем данного типа SGI Origin2000, 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.

Мультикомпьютеры (системы с распределенной памятью) уже не обеспечивают общий доступ ко всей имеющейся в системах памяти (*no-remote memory access or NORMA*). Данный подход используется при построении двух важных типов многопроцессорных вычислительных систем - *массивно-параллельных систем* (*massively parallel processor or MPP*) и *кластеров* (*clusters*). Среди представителей первого типа систем - IBM RS/6000 SP2, Intel PARAGON/ASCI Red, транспьютерные системы Parsytec и др.; примерами кластеров являются, например, системы AC3 Velocity и NCSA/NT Supercluster.

Следует отметить чрезвычайно быстрое развитие **кластерного типа** многопроцессорных вычислительных систем - современное состояние данного подхода отражено, например, в обзоре, подготовленном под редакцией Barker (2000). Под кластером обычно понимается (см., например, Xu and Hwang (1998), Pfister (1998)) множество отдельных компьютеров, объединенных в сеть, для которых при помощи специальных аппаратно-программных средств обеспечивается возможность унифицированного управления (single system image), надежного функционирования (availability) и эффективного использования (performance). Кластеры могут быть образованы на базе уже существующих у потребителей отдельных компьютеров либо же сконструированы из типовых компьютерных элементов, что обычно не требует значительных финансовых затрат. Применение кластеров может также в некоторой степени снизить проблемы, связанные с разработкой параллельных алгоритмов и программ, поскольку

повышение вычислительной мощности отдельных процессоров позволяет строить кластеры из сравнительно небольшого количества (несколько десятков) отдельных компьютеров (lowly parallel processing). Это приводит к тому, что для параллельного выполнения в алгоритмах решения вычислительных задач достаточно выделять только крупные независимые части расчетов (coarse granularity), что, в свою очередь, снижает сложность построения параллельных методов вычислений и уменьшает потоки передаваемых данных между компьютерами кластера. Вместе с этим следует отметить, что организации взаимодействия вычислительных узлов кластера при помощи передачи сообщений обычно приводит к значительным временным задержкам, что накладывает дополнительные ограничения на тип разрабатываемых параллельных алгоритмов и программ.

В завершении обсуждаемой темы можно отметить, что существуют и другие способы классификации вычислительных систем (достаточно полный обзор подходов представлен в [22], см. также материалы сайта http://www.parallel.ru/computers/taxonomy/); при рассмотрении данной темы параллельных вычислений рекомендуется обратить внимание на способ структурной нотации для описания архитектуры ЭВМ, позволяющий с высокой степенью точности описать многие характерные особенности компьютерных систем.

1.3. Характеристика типовых схем коммуникации в многопроцессорных вычислительных системах

При организации параллельных вычислений в МВС для организации взаимодействия, синхронизации и взаимоисключения параллельно выполняемых процессов используется передача данных между процессорами вычислительной среды. Временные задержки при передаче данных по линиям связи могут оказаться существенными (по сравнению с быстродействием процессоров) и, как результат, *коммуникационная трудоемкость* алгоритма оказывает существенное влияние на выбор параллельных способов решения задач.

Структура линий коммутации между процессорами вычислительной системы (*топология сети передачи данных*) определяется, как правило, с учетом возможностей эффективной технической реализации; немаловажную роль при выборе структуры сети играет и анализ интенсивности информационных потоков при параллельном решении наиболее распространенных вычислительных задач. К числу типовых топологий обычно относят следующие схемы коммуникации процессоров (см. рис. 1.1):

- **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 все процессоры имеют линии связи с некоторым управляющим процессором; данная топология является эффективной, например, при организации централизованных схем параллельных вычислений;

- **решетка** (*mesh*) - система, в которой граф линий связи образует прямоугольную сетку (обычно двух- или трех- мерную); подобная топология может быть достаточно просто реализована и, кроме того, может быть эффективно используема при параллельном выполнении многих численных алгоритмов (например, при реализации методов анализа математических моделей, описываемых дифференциальными уравнениями в частных производных);

• **гиперкуб** (*hypercube*) - данная топология представляет частный случай структуры 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:

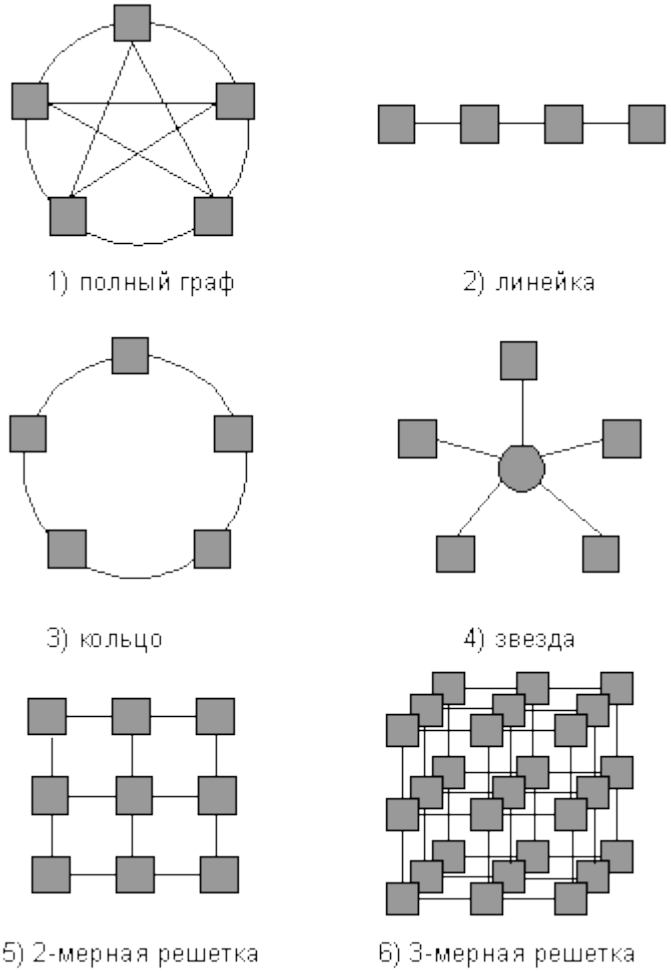

Рис. 1.2. Примеры топологий многопроцессорных вычислительных систем

- 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*).

Дополнительная информация по топологиям МВС может быть получена, например, в [9, 22 - 23, 29, 31]; при рассмотрении вопроса следует учесть, что схема линий передачи данных может реализовываться на аппаратном уровне, а может быть обеспечена на основе имеющейся физической топологии при помощи соответствующего программного обеспечения. Введение *виртуальных* (программно-реализуемых) топологий способствует мобильности разрабатываемых параллельных программ и снижает затраты на программирование.

1.4. Высокопроизводительный вычислительный кластер ННГУ

Для проведения вычислительных экспериментов использовался вычислительный кластер Нижегородского университета, оборудование для которого было передано в рамках Академической программы Интел в 2001 г. В состав кластера входит (см. рис. 1.3):

- 2 вычислительных сервера, каждый из которых имеет 4 процессора Intel Pentium III 700 Мгц, 512 MB RAM, 10 GB HDD, 1 Гбит Ethernet card;
- 12 вычислительных серверов, каждый из которых имеет 2 процессора Intel Pentium III 1000 Мгц, 256 MB RAM, 10 GB HDD, 1 Гбит Ethernet card;
- 12 рабочих станций на базе процессора Intel Pentium 4 1300 Мгц, 256 MB RAM, 10 GB HDD, CD-ROM, монитор 15", 10/100 Fast Ethernet card.

Следует отметить, что в результате передачи подобного оборудования Нижегородский госуниверситет оказался первым вузом в Восточной Европе, оснащенным ПК на базе новейшего процессора INTEL® PENTIUM® 4. Подобное достижение является дополнительным подтверждением складывающегося плодотворного сотрудничества ННГУ и корпорации Интел.

Важной отличительной особенностью кластера является его неоднородность (*гетерогенность*). В состав кластера входят рабочие места, оснащенные новейшими процессорами Intel Pentium 4 и соединенные относительно медленной сетью (100 Мбит), а также вычислительные 2- и 4- процессорные сервера, обмен данными между которыми выполняется при помощи быстрых каналов передачи данных (1000 Мбит). В результате кластер может использоваться не только для решения сложных вычислительно-трудоемких задач, но также и для проведения различных экспериментов по исследованию многопроцессорных кластерных систем и параллельных методов решения научно-технических задач.

В качестве системной платформы для построения кластера выбраны современные операционные системы семейства Microsoft Windows (для проведения отдельных экспериментов имеется возможность использования ОС Unix). Выбор такого решения определяется рядом причин, в числе которых основными являются следующие моменты:

- операционные системы семейства Microsoft Windows (так же как и ОС Unix) широко используются для построения кластеров; причем, если раньше применение ОС Unix для этих целей было преобладающим системным решением, в настоящее время тенденцией является увеличение числа создаваемых кластеров вод управлением ОС Microsoft Windows (см., например, www.tc.cornell.edu/ac3/, www.windowclusters.org и др.);
- разработка прикладного программного обеспечения выполняется преимущественно с использованием ОС Microsoft Windows;

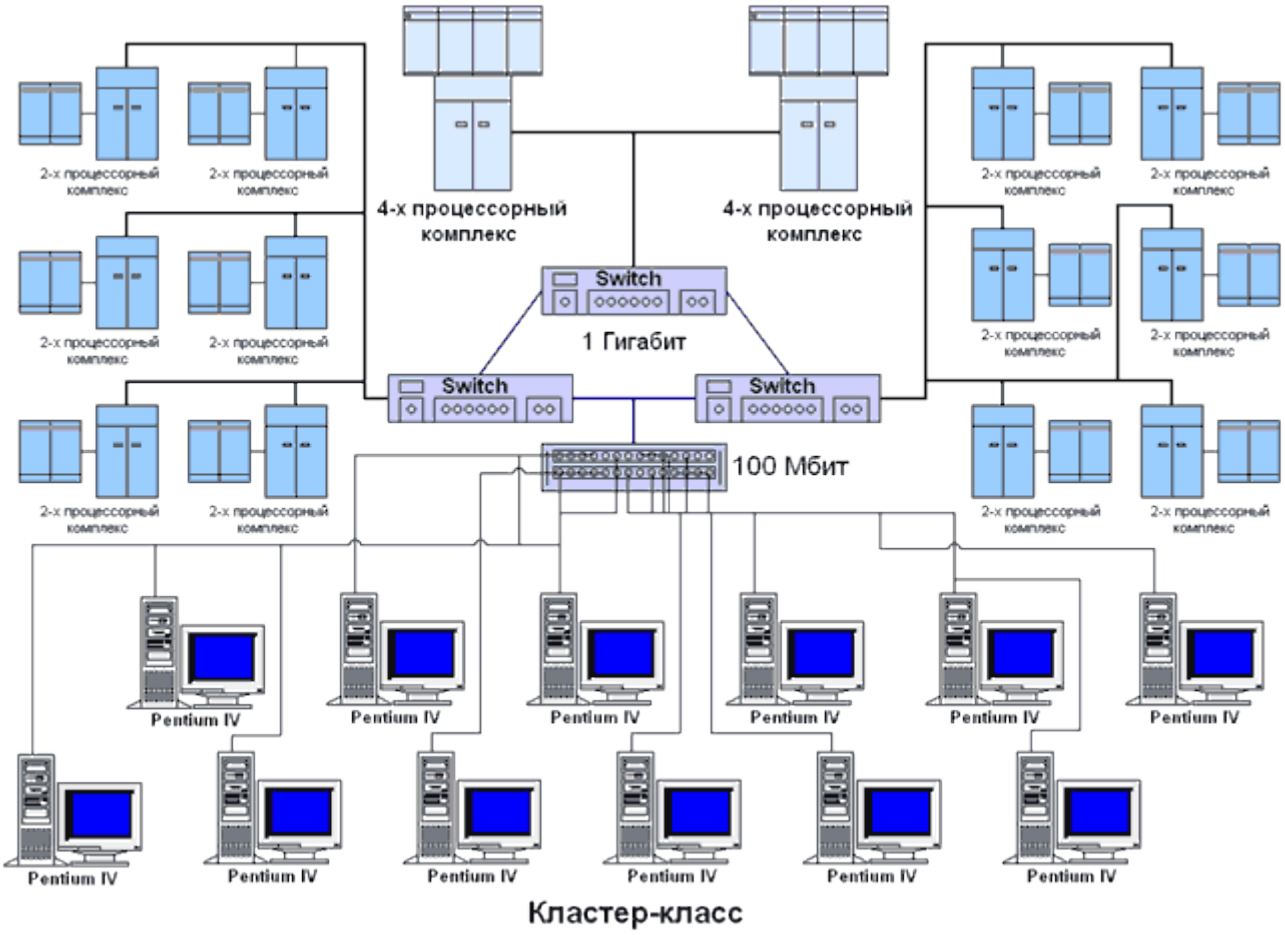

Рис. 1.3. Структура вычислительного кластера Нижегородского университета
(Нажмите на изображение для его увеличения)

• корпорация Microsoft проявила заинтересованность в создании подобного кластера и передала в ННГУ для поддержки работ все необходимое программное обеспечение (ОС MS Windows 2000 Professional, ОС MS Windows 2000 Advanced Server и др.).

В результате принятых решений программное обеспечение кластера является следующим:

• вычислительные сервера работают под управлением ОС Microsoft Windows 2000 Advanced Server; на рабочих местах разработчиков установлена ОС Microsoft Windows 2000 Professional;

• в качестве сред разработки используются Microsoft Visual Studio 6.0; для выполнения исследовательских экспериментов возможно использование компилятора Intel C++ Compiler 5.0, встраиваемого в среду Microsoft Visual Studio;

• на рабочих местах разработчиков установлены библиотеки:

- Plapack 3.0 (см. www.cs.utexas.edu/users/plapack);
- MKL (см. developer.intel.com/software/products/mkl/index.htm);

• в качестве средств передачи данных между процессорами установлены две реализации стандарта MPI:

- Argonne MPICH (www-unix.mcs.anl.gov/mpi/MPICH/);
- MP-MPICH (www.lfbs.rwth-aachen.de/~joachim/MP-MPICH.html).

• в опытной эксплуатации находится система разработки параллельных программ DVM (см. www.keldysh.ru/dvm/).

2. Моделирование и анализ параллельных вычислений

При разработке параллельных алгоритмов решения задач вычислительной математики принципиальным моментом является анализ эффективности использования параллелизма, состоящий обычно в оценке получаемого *ускорения* процесса вычисления (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 получения 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 в 4 разделе пособия.

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" [18] (дополнительная информация по моделированию параллельных вычислений может быть получена в [3, 16, 23, 26, 28]). 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). Анализ коммуникационной трудоемкости параллельных алгоритмов выполняется в 3 разделе пособия.

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

$G = (V, R)$,

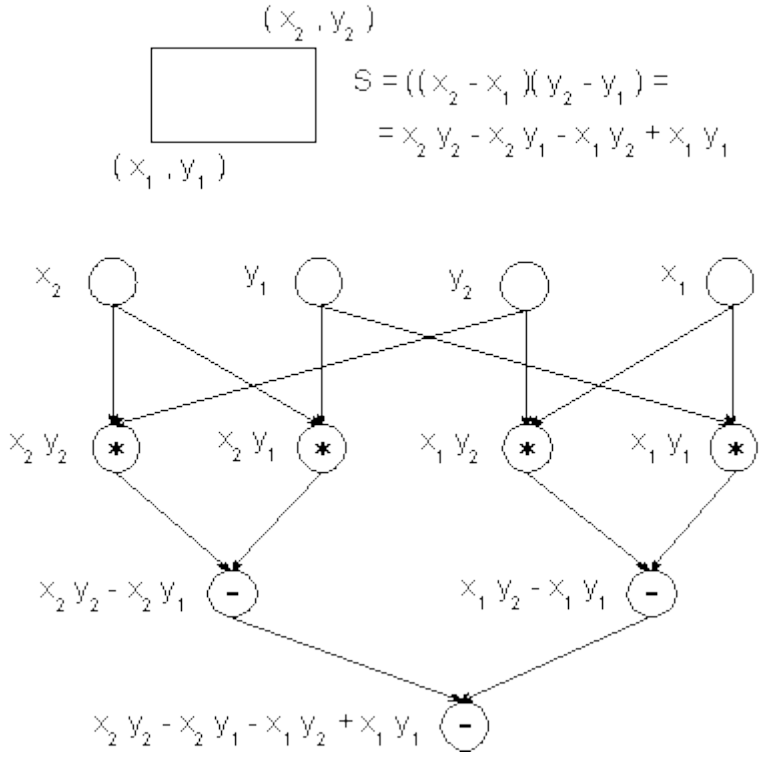

Рис. 2.1. Пример вычислительной модели алгоритма в виде графа "операции-операнды"

где $V = \{1, \ldots, |V|\}$ есть множество вершин графа, представляющее 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 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 разными 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 $\overline{V}$ 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 выполнены сначала все операции умножения, а затем первые две операции вычитания). Возможный способ описания параллельного выполнения алгоритма может состоять в следующем [18].

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

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

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

Модель вычислительной схемы 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

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

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 возможной параллельности можно определить оценку наиболее быстрого исполнения алгоритма

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

Оценку T_∞ 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 проблемой при анализе параллельных алгоритмов, поскольку применяется для определения эффекта использования параллельности (ускорения времени решения задачи). Очевидно

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

где $|\overline{V}|$, 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

$$T_1 = \min_{G} T_1(G),$$

то получаемые при использовании такой оценки показатели ускорения будут характеризовать эффективность распараллеливания выбранного алгоритма. Для оценки эффективности параллельного решения исследуемой задачи вычислительной математики величину T_1 следует определять с учетом всех возможных последовательных алгоритмов

$$T_1^* = \min T_1$$

(эффективный параллельный алгоритм может не совпадать с наилучшим последовательным методом при исполнении на одном процессоре).

Приведем без доказательства теоретические положения, характеризующие свойства оценок времени выполнения параллельного алгоритма [18].

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 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.

$$\forall q = cp, \quad c > 0 \Rightarrow T_p \le cT_q.$$

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, сопоставимым с минимально возможным временем T_∞, можно достичь при количестве процессоров порядка $p \sim T_1 / T_\infty$, а именно,

$$p \ge T_1 / T_\infty \Rightarrow T_p \le 2T_\infty.$$

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

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

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_∞ есть расписание для достижения минимально возможного времени выполнения T_∞. Для каждой итерации τ, $0 \le \tau \le T_\infty$, выполнения расписания H_∞ обозначим через n_τ количество операций, выполняемых в ходе итерации τ. Расписание выполнения алгоритма с использованием p процессоров может быть построено следующим образом. Выполнение алгоритма разделим на T_∞ шагов; на каждом шаге τ следует выполнить все операции, которые выполнялись на итерации τ расписания H_∞. Выполнение этих операций может быть выполнено не более, чем за $\lceil n_\tau / p \rceil$ итераций при использовании p процессоров. Как результат, время выполнения алгоритма T_p может быть оценено следующим образом:

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

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

2.4. Показатели эффективности параллельного алгоритма

Ускорение, получаемое при использовании параллельного алгоритма для p процессоров, по сравнению с последовательным вариантом выполнения вычислений определяется

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

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 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).

Как следует из приведенных соотношений, в наилучшем случае $S_p(n) = p$ и $E_p(n) = 1$. В **4 разделе** данные показатели будут определены для ряда рассмотренных параллельных алгоритмов для решения типовых задач вычислительной математики.

3. Оценка коммуникационной трудоемкости параллельных алгоритмов

Как уже отмечалось ранее, временные задержки при передаче данных по каналам связи для организации взаимодействия раздельно-функционирующих процессов могут в значительной степени определять эффективность параллельных вычислений. Данный раздел посвящен вопросам анализа информационных потоков, возникающих при выполнении параллельных алгоритмов. В разделе определяются показатели, характеризующие свойства топологий коммуникационных сетей, дается общая характеристика механизмов передачи данных, проводится анализ трудоемкости основных операций обмена информацией,

рассматриваются методы логического представления структуры МВС. Более подробно изучаемый в данном разделе пособия учебный материал излагается в [18, 23, 28].

3.1. Характеристики топологии сети передачи данных

В качестве основных характеристик топологии сети передачи данных наиболее широко используется следующий ряд показателей:

- **диаметр** - показатель, определяемый как максимальное расстояние между двумя процессорами сети (под расстоянием обычно понимается величина кратчайшего пути между процессорами); данная величина может охарактеризовать максимально-необходимое время для передачи данных между процессорами, поскольку время передачи обычно прямо пропорционально длине пути;
- **связность** (*connectivity*) - показатель, характеризующий наличие разных маршрутов передачи данных между процессорами сети; конкретный вид данного показателя может быть определен, например, как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на две несвязные области;
- **ширина бинарного деления** (*bisection width*) - показатель, определяемый как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на две несвязные области одинакового размера;
- **стоимость** - показатель, который может быть определен, например, как общее количество линий передачи данных в многопроцессорной вычислительной системе.

Для сравнения в таблице 3.1 приводятся значения перечисленных показателей для различных топологий сети передачи данных.

Таблица 3.1. Характеристики топологий сети передачи данных (p - количество процессоров)

Топология	Диаметр	Ширина бисекции	Связность	Стоимость
Полный граф	1	$p^2 / 4$	$p - 1$	$p\,(p-1) / 2$
Звезда	2	1	1	$p - 1$
Полное двоичное дерево	$2 \log ((p+1)/2)$	1	1	$p - 1$
Линейка	$p - 1$	1	1	$p - 1$
Кольцо	$\lfloor p / 2 \rfloor$	2	2	p
Решетка ($N = 2$)	$2(\sqrt{p} - 1)$	$\sqrt{p}$	2	$2\,(p - \sqrt{p})$
Решетка-тор ($N = 2$)	$2\lfloor \sqrt{p} / 2 \rfloor$	$2\sqrt{p}$	4	$2p$
Гиперкуб	$\log p$	$p / 2$	$\log p$	$(p \log p) / 2$

3.2. Общая характеристика механизмов передачи данных

Алгоритмы маршрутизации

Алгоритмы маршрутизации определяют путь передачи данных от процессора-источника сообщения до процессора, к которому сообщение должно быть доставлено. Среди возможных способов решения данной задачи различают:

- *оптимальные*, определяющие всегда наикратчайшие пути передачи данных, и *неоптимальные* алгоритмы маршрутизации;

- *детерминированные* и *адаптивные* методы выбора маршрутов (адаптивные алгоритмы определяют пути передачи данных в зависимости от существующей загрузки коммуникационных каналов).

К числу наиболее распространенных оптимальных алгоритмов относится класс *методов покоординатной маршрутизации* (*dimension-ordered routing*), в которых поиск путей передачи данных осуществляется поочередно для каждой размерности топологии сети коммуникации. Так, для двумерной решетки такой подход приводит к маршрутизации, при которой передача данных сначала выполняется по одному направлению (например, по горизонтали до достижения вертикали процессоров, в которой располагается процессор назначения), а затем данные передаются вдоль другого направления (данная схема известна под названием *алгоритма XY-маршрутизации*).

Для гиперкуба покоординатная схема маршрутизации может состоять, например, в циклической передаче данных процессору, определяемому первой различающейся битовой позицией в номерах процессоров, на котором сообщение располагается в данный момент времени и на который сообщение должно быть передано.

Методы передачи данных

Время передачи данных между процессорами определяет коммуникационную составляющую (communication latency) длительности выполнения параллельного алгоритма в многопроцессорной вычислительной системе. Основной набор параметров, описывающих время передачи данных, состоит из следующего ряда величин:

- **время начальной подготовки** ($t_н$) характеризует длительность подготовки сообщения для передачи, поиска маршрута в сети и т.п.;
- **время передачи служебных данных** (t_c) между двумя соседними процессорами (т.е. для процессоров, между которыми имеется физический канал передачи данных); к служебным данным может относиться заголовок сообщения, блок данных для обнаружения ошибок передачи и т.п.;
- **время передачи одного слова данных** по одному каналу передачи данных ($t_к$); длительность подобной передачи определяется полосой пропускания коммуникационных каналов в сети.

К числу наиболее распространенных методов передачи данных относятся следующие два основные способа коммуникации [23]. Первый из них ориентирован на *передачу сообщений* (МПС) как неделимых (атомарных) блоков информации (*store-and-forward routing* or *SFR*). При таком подходе процессор, содержащий сообщение для передачи, готовит весь объем данных для передачи, определяет процессор, которому следует направить данные, 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 $t_{пд}$ для метода передачи сообщения размером m по маршруту длиной l определяется выражением

$$t_{пд} = t_н + (mt_к + t_c)l \ .$$

При достаточно длинных сообщениях временем передачи служебных данных можно пренебречь и выражение для времени передачи данных может быть записано в более простом виде

$$t_{пд} = t_н + mt_к l \ .$$

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*) принимающий процессор может осуществлять пересылку данных по

дальнейшему маршруту непосредственно сразу после приема очередного пакета, не дожидаясь завершения приема данных всего сообщения. Время пересылки данных при использовании метода передачи пакетов будет определяться выражением

$$t_{пд} = t_н + mt_к + t_c l \ .$$

Сравнивая полученные выражения, можно заметить, что в большинстве случаев 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. С другой стороны, реализация пакетного метода требует разработки более сложного аппаратного и программного обеспечения сети, может увечить накладные расходы (время подготовки и время передачи служебных данных); при передаче пакетов возможно возникновения конфликтных ситуаций (дедлоков).

3.3. Анализ трудоемкости основных операций передачи данных

При всем разнообразии выполняемых операций передачи данных при параллельных способах решения сложных научно-технических задач определенные процедуры взаимодействия процессоров сети могут быть отнесены к числу основных коммуникационных действий, которые или наиболее широко распространены в практике параллельных вычислений, или к которым могут быть сведены многие другие процессы приема-передачи сообщений. Важно отметить также, что в рамках подобного базового набора для большинства операций коммуникации существуют процедуры, обратные по действию исходным операциям (так, например, операции передачи данных от одного процессора всем имеющимся процессорам сети соответствует операция приема в одном процессоре сообщений от всех остальных процессоров). Как результат, рассмотрение коммуникационных процедур целесообразно выполнять попарно, поскольку во многих случаях алгоритмы выполнения прямой и обратной операций могут быть получены исходя из общих положений.

Рассмотрение основных операций передачи данных в данном разделе будет осуществляться на примере таких топологий сети, как кольцо, двумерной решетки и гиперкуба. Для двумерной решетки будет предполагаться также, что между граничными процессорами в строках и столбцах решетки имеются каналы передачи данных (т.е. топология сети представляет из себя тор). Как и ранее, величина m будет означать размер сообщения в словах, значение p определяет количество процессоров в сети, а переменная N задает размерность топологии гиперкуба.

Передача данных между двумя процессорами сети

Трудоемкость данной коммуникационной операции может быть получена путем подстановки длины максимального пути (диаметра сети - см. табл. 3.1) в выражения для времени передачи данных при разных методах коммуникации (см. п. 3.2).

Таблица 3.2. Время передачи данных между двумя процессорами

Топология	Передача сообщений	Передача пакетов
Кольцо	$t_н + mt_к\lfloor p/2 \rfloor$	$t_н + mt_к + t_c\lfloor p/2 \rfloor$
Решетка-тор	$t_н + 2mt_к\lfloor \sqrt{p}/2 \rfloor$	$t_н + mt_к + 2t_c\lfloor \sqrt{p}/2 \rfloor$
Гиперкуб	$t_н + mt_к \log p$	$t_н + mt_к + t_c \log p$

Передача данных от одного процессора всем остальным процессорам сети

Операция передачи данных (одного и того же сообщения) от одного процессора всем остальным процессорам сети (*one-to-all broadcast* or *single-node broadcast*) является одним из наиболее часто выполняемых коммуникационных действий; двойственная операция передачи -

прием на одном процессоре сообщений от всех остальных процессоров сети (*single-node accumulation*). Подобные операции используются, в частности, при реализации матрично-векторного произведения, решении систем линейных уравнений при помощи метода Гаусса, поиска кратчайших путей и др.

Простейший способ реализации операции рассылки состоит в ее выполнении как последовательности попарных взаимодействий процессоров сети. Однако при таком подходе большая часть пересылок является избыточной и возможно применение более эффективных алгоритмов коммуникации. Изложение материала будет проводиться сначала для метода передачи сообщений, затем - для пакетного способа передачи данных (см. п. 3.2).

1. **Передача сообщений**. Для **кольцевой топологии** процессор-источник рассылки может инициировать передачу данных сразу двум своим соседям, которые, в свою очередь, приняв сообщение, организуют пересылку далее по кольцу. Трудоемкость выполнения операции рассылки в этом случае будет определяться соотношением

$$t_{пд} = (t_н + mt_к)\lceil p / 2 \rceil .$$

Для топологии типа **решетки-тора** алгоритм рассылки может быть получен из способа передачи данных, примененного для кольцевой структуры сети. Так, рассылка может быть выполнена в виде двухэтапной процедуры. На первом этапе организуется передача сообщения всем процессорам сети, располагающимся на той же горизонтали решетки, что и процессор-инициатор передачи; на втором этапе процессоры, получившие копию данных на первом этапе, рассылают сообщения по своим соответствующим вертикалям. Оценка длительности операции рассылки в соответствии с описанным алгоритмом определяется соотношением

$$t_{пд} = 2(t_н + mt_к)\lceil \sqrt{p} / 2 \rceil .$$

Для **гиперкуба** рассылка может быть выполнена в ходе N-этапной процедуры передачи данных. На первом этапе процессор-источник сообщения передает данные одному из своих соседей (например, по первой размерности) - в результате после первого этапа имеется два процессора, имеющих копию пересылаемых данных (данный результат можно интерпретировать также как разбиение исходного гиперкуба на два таких одинаковых по размеру гиперкуба размерности $N - 1$, что каждый из них имеет копию исходного сообщения). На втором этапе два процессора, задействованные на первом этапе, пересылают сообщение своим соседям по второй размерности и т.д. В результате такой рассылки время операции оценивается при помощи выражения

$$t_{пд} = (t_н + mt_к) \log p .$$

Сравнивая полученные выражения для длительности выполнения операции рассылки, можно отметить, что наилучшие показатели имеет топология типа гиперкуба; более того, можно показать, что данный результат является наилучшим для выбранного способа коммуникации с помощью передачи сообщений.

2. **Передача пакетов**. Для топологии типа **кольца** алгоритм рассылки может быть получен путем логического представления кольцевой структуры сети в виде гиперкуба. В результате на этапе рассылки процессор-источник сообщения передает данные процессору, находящемуся на расстоянии $p / 2$ от исходного процессора. Далее, на втором этапе оба процессора, уже имеющие рассылаемые данные после первого этапа, передают сообщения процессорам, находящиеся на расстоянии $p / 4$ и т.д. Трудоемкость выполнения операции рассылки при таком методе передачи данных определяется соотношением

$$t_{пд} = \sum_{i=1}^{\log p} (t_н + mt_к + t_c\, p / 2^i) = (t_н + mt_к) \log p + t_c(p - 1)$$

(как и ранее, при достаточно больших сообщениях, временем передачи служебных данных можно пренебречь).

Для топологии типа **решетки-тора** алгоритм рассылки может быть получен из способа передачи данных, примененного для кольцевой структуры сети, в соответствии с тем же способом обобщения, что и в случае использования метода передачи сообщений. Получаемый в результате такого обобщения алгоритм рассылки характеризуется следующим соотношением для оценки времени выполнения:

$$t_{n\partial} = (t_н + mt_к) \log p + 2t_c(\sqrt{p} - 1) .$$

Для **гиперкуба** алгоритм рассылки (и, соответственно, временные оценки длительности выполнения) при передаче пакетов не отличается от варианта для метода передачи сообщений.

Передача данных от всех процессоров всем процессорам сети

Операция передачи данных от всех процессоров всем процессорам сети (*all-to-all broadcast* or *multinode broadcast*) является естественным обобщением одиночной операции рассылки; двойственная операция передачи - прием сообщений на каждом процессоре от всех процессоров сети (*multinode accumulation*). Подобные операции широко используются, например, при реализации матричных вычислений.

Возможный способ реализации операции множественной рассылки состоит в выполнении соответствующего набора операций одиночной рассылки. Однако такой подход не является оптимальным для многих топологий сети, поскольку часть необходимых операций одиночной рассылки потенциально может быть выполнена параллельно. Как и ранее, учебный материал будет рассматриваться раздельно для разных методов передачи данных (см. п. 3.2).

1. **Передача сообщений**. Для **кольцевой топологии** каждый процессор может инициировать рассылку своего сообщения одновременно (в каком-либо выбранном направлении по кольцу). В любой момент времени каждый процессор выполняет прием и передачу данных; завершение операции множественной рассылки произойдет через $(p - 1)$ цикл передачи данных. Длительность выполнения операции рассылки оценивается соотношением:

$$t_{n\partial} = (t_н + mt_к)(p - 1) .$$

Для топологии типа **решетки-тора** множественная рассылка сообщений может быть выполнена при помощи алгоритма, получаемого обобщением способа передачи данных для кольцевой структуры сети. Схема обобщения состоит в следующем. На первом этапе организуется передача сообщений раздельно по всем процессорам сети, располагающимся на одних и тех же горизонталях решетки (в результате на каждом процессоре одной и той же горизонтали формируются укрупненные сообщения размера $m\sqrt{p}$, объединяющие все сообщения горизонтали). Время выполнения этапа

$$t'_{n\partial} = (t_н + mt_к)(\sqrt{p} - 1) .$$

На втором этапе рассылка данных выполняется по процессорам сети, образующим вертикали решетки. Длительность этого этапа

$$t''_{n\partial} = (t_н + m\sqrt{p}\, t_к)(\sqrt{p} - 1) .$$

Как результат, общая длительность операции рассылки определяется соотношением:

$$t_{n\partial} = 2t_н(\sqrt{p} - 1) + mt_к(p - 1) .$$

Для **гиперкуба** алгоритм множественной рассылки сообщений может быть получен путем обобщения ранее описанного способа передачи данных для топологии типа решетки на размерность гиперкуба N. В результате такого обобщения схема коммуникации состоит в следующем. На каждом этапе i, $1 \le i \le N$, выполнения алгоритма функционируют все

процессоры сети, которые обмениваются своими данными со своими соседями по i размерности и формируют объединенные сообщения. Время операции рассылки может быть получено при помощи выражения

$$t_{n\partial} = \sum_{i=1}^{\log p} (t_н + 2^{i-1} m t_к) = t_н \log p + m t_к (p - 1)$$

2. **Передача пакетов**. Применение более эффективного для кольцевой структуры и топологии типа решетки-тора метода передачи данных не приводит к какому-либо улучшению времени выполнения операции множественной рассылки, поскольку обобщение алгоритмов выполнения операции одиночной рассылки на случай множественной рассылки приводит к перегрузке каналов передачи данных (т.е. к существованию ситуаций, когда в один и тот же момент времени для передачи по одной и той линии передачи имеется несколько ожидающих пересылки пакетов данных). Перегрузка каналов приводит к задержкам при пересылках данных, что и не позволяет проявиться всем преимуществам метода передачи пакетов.

Возможным широко распространенным примером операции множественной рассылки является задача **редукции** (*reduction*), определяемая в общем виде, как процедура выполнения той или иной обработки получаемых на каждом процессоре данных (в качестве примера такой задачи может быть рассмотрена проблема вычисления суммы значений, находящихся на разных процессорах, и рассылки полученной суммы по всем процессорам сети). Способы решения задачи редукции могут состоять в следующем:

- непосредственный подход заключается в выполнении операции множественной рассылки и последующей затем обработке данных на каждом процессоре в отдельности;
- более эффективный алгоритм может быть получен в результате применения операции одиночного приема данных на отдельном процессоре, выполнения на этом процессоре действий по обработке данных, и рассылке полученного результата обработки всем процессорам сети;
- наилучший же способ решения задачи редукции состоит в совмещении процедуры множественной рассылки и действий по обработке данных, когда каждый процессор сразу же после приема очередного сообщения реализует требуемую обработку полученных данных (например, выполняет сложение полученного значения с имеющейся на процессоре частичной суммой). Время решения задачи редукции при таком алгоритме реализации в случае, например, когда размер пересылаемых данных имеет единичную длину ($m = 1$) и топология сети имеет структуру гиперкуба, определяется выражением

  $$t_{n\partial} = (t_н + t_к) \log p \ .$$

Другим типовым примером используемости операции множественной рассылки является **задача нахождения частных сумм** последовательности значений S_i (в зарубежной литературе эта задача известна под названием *prefix sum problem*)

$$S_k = \sum_{i=1}^{k} x_i, \ 1 \le k \le p$$

(будем предполагать, что количество значений совпадает с количеством процессоров, значение x_i располагается на i процессоре и результат S_k должен получаться на процессоре с номером k).

Алгоритм решения данной задачи также может быть получен при помощи конкретизации общего способа выполнения множественной операции рассылки, когда процессор выполняет суммирование полученного значения (но только в том случае, если процессор-отправитель значения имеет меньший номер, чем процессор-получатель).

Обобщенная передача данных от одного процессора всем остальным процессорам сети

Общий случай передачи данных от одного процессора всем остальным процессорам сети состоит в том, что все рассылаемые сообщения являются различными (*one-to-all personalized communication* or *single-node scatter*). Двойственная операция передачи для данного типа взаимодействия процессоров - обобщенный прием сообщений (*single-node gather*) на одном процессоре от всех остальных процессоров сети (отличие данной операции от ранее рассмотренной процедуры сборки данных на одном процессоре (*single-node accumulation*) состоит в том, что обобщенная операция сборки не предполагает какого-либо взаимодействия сообщений (типа редукции) в процессе передачи данных).

Трудоемкость операции обобщенной рассылки сопоставима со сложностью выполнения процедуры множественной передачи данных. Процессор-инициатор рассылки посылает каждому процессору сети сообщение размера m и, тем самым, нижняя оценка длительности выполнения операции характеризуется величиной $mt_k(p-1)$.

Проведем более подробный анализ трудоемкости обобщенной рассылки для случая топологии типа **гиперкуб**. Возможный способ выполнения операции состоит в следующем. Процессор-инициатор рассылки передает половину своих сообщений одному из своих соседей (например, по первой размерности) - в результате, исходный гиперкуб становится разделенным на два гиперкуба половинного размера, в каждом из которых содержится ровно половина исходных данных. Далее действия по рассылке сообщений могут быть повторены и общее количество повторений определяется исходной размерностью гиперкуба. Длительность операции обобщенной рассылки может быть охарактеризована соотношением:

$$t_{по} = t_н \log p + mt_k(p-1)$$

(как и отмечалась выше, трудоемкость операции совпадает с длительностью выполнения процедуры множественной рассылки).

Обобщенная передача данных от всех процессоров всем процессорам сети

Обобщенная передача данных от всех процессоров всем процессорам сети (*total exchange*) представляет собой наиболее общий случай коммуникационных действий. Необходимость в выполнении подобных операций возникает в параллельных алгоритмах быстрого преобразования Фурье, транспонирования матриц и др.

Выполним краткую характеристику возможных способов выполнения обобщенной множественной рассылки для разных методов передачи данных (см. п. 3.2).

1. **Передача сообщений**. Общая схема алгоритма для **кольцевой топологии** состоит в следующем. Каждый процессор производит передачу всех своих исходных сообщений своему соседу (в каком-либо выбранном направлении по кольцу). Далее процессоры осуществляют прием направленных к ним данных, затем среди принятой информации выбирают свои сообщения, после чего выполняет дальнейшую рассылку оставшейся части данных. Длительность выполнения подобного набора передач данных оценивается при помощи выражения:

$$t_{по} = (t_н + \tfrac{1}{2}\, mpt_k)(p-1)$$

Способ получения алгоритма рассылки данных для топологии типа **решетки-тора** является тем же самым, что и в случае рассмотрения других коммуникационных операций. На первом этапе организуется передача сообщений раздельно по всем процессорам сети, располагающимся на одних и тех же горизонталях решетки (каждому процессору по горизонтали передаются только те исходные сообщения, что должны быть направлены процессорам соответствующей вертикали решетки); после завершения этапа на каждом процессоре собираются p сообщений, предназначенных для рассылки по одной из вертикалей решетки. На втором этапе рассылка данных выполняется по процессорам сети, образующим

вертикали решетки. Общая длительность всех операций рассылок определяется соотношением

$$t_{нд} = 2(t_н + mpt_k)(\sqrt{p} - 1)$$

Для **гиперкуба** алгоритм обобщенной множественной рассылки сообщений может быть получен путем обобщения способа выполнения операции для топологии типа решетки на размерность гиперкуба N. В результате такого обобщения схема коммуникации состоит в следующем. На каждом этапе i, $1 \le i \le N$, выполнения алгоритма функционируют все процессоры сети, которые обмениваются своими данными со своими соседями по i размерности и формируют объединенные сообщения. При организации взаимодействия двух соседей канал связи между ними рассматривается как связующий элемент двух равных по размеру подгиперкубов исходного гиперкуба, и каждый процессор пары посылает другому процессору только те сообщения, что предназначены для процессоров соседнего подгиперкуба. Время операции рассылки может быть получено при помощи выражения:

$$t_{нд} = (t_н + ½\ mpt_k) \log p$$

(кроме затрат на пересылку, каждый процессор выполняет $mp \log p$ операций по сортировке своих сообщений перед обменом информацией со своими соседями).

2. **Передача пакетов**. Как и в случае множественной рассылки, применение метода передачи пакетов не приводит к улучшению временных характеристик для операции обобщенной множественной рассылки. Рассмотрим для примера более подробно выполнение данной коммуникационной операции для сети с топологией типа **гиперкуб**. В этом случае рассылка может быть выполнена за $p - 1$ последовательных итераций. На каждой итерации все процессоры разбиваются на взаимодействующие пары процессоров, причем это разбиение на пары может быть выполнено таким образом, чтобы передаваемые между разными парами сообщения не использовали одни и те же пути передачи данных. Как результат, общая длительность операции обобщенной рассылки может быть определена в соответствии с выражением:

$$t_{нд} = (t_н + mt_k)(p - 1) + ½\ t_c\ p \log p$$

Циклический сдвиг

Частный случай обобщенной множественной рассылки есть процедура перестановки (permutation), представляющая собой операцию перераспределения информации между процессорами сети, в которой каждый процессор передает сообщение некоторому определенному другому процессору сети. Конкретный вариант перестановки - циклический q-сдвиг (circular q-shift), при котором каждый процессор i, $1 \le i \le p$, передает данные процессору с номером $(i + q) \bmod p$. Подобная операция сдвига используется, например, при организации матричных вычислений.

Поскольку выполнение циклического сдвига для кольцевой топологии может быть обеспечено при помощи простых алгоритмов передачи данных, рассмотрим возможные способы выполнения данной коммуникационной операции только для топологий решетки-тора и гиперкуба при разных методах передачи данных (см. п. 3.2).

1. **Передача сообщений**. Общая схема алгоритма циклического сдвига для топологии типа **решетки-тора** состоит в следующем. Пусть процессоры перенумерованы по строкам решетки от 0 до $p - 1$. На первом этапе организуется циклический сдвиг с шагом $q \bmod \sqrt{p}$ по каждой строке в отдельности (если при реализации такого сдвига сообщения передаются через правые границы строк, то после выполнения каждой такой передачи необходимо осуществить компенсационный сдвиг вверх на 1 для процессоров первого столбца решетки). На втором этапе реализуется циклический сдвиг вверх с шагом $\lfloor q / \sqrt{p} \rfloor$ для каждого столбца решетки. Общая длительность всех операций рассылок определяется соотношением

$t_{nд} = (t_н + mt_k)(2\lfloor\sqrt{p}\ / 2\rfloor + 1)$

Для **гиперкуба** алгоритм циклического сдвига может быть получен путем логического представления топологии гиперкуба в виде кольцевой структуры. Для получения такого представления установим взаимно-однозначное соответствие между вершинами кольца и гиперкуба. Необходимое соответствие может быть получено, например, при помощи известного кода Грея, который можно использовать для определения процессоров гиперкуба, соответствующих конкретным вершинам кольца. Более подробное изложение механизма установки такого соответствия осуществляется в п. 3.4; для наглядности на рис. 3.1 приводится вид гиперкуба для размерности $N = 3$ с указанием для каждого процессора гиперкуба соответствующей вершины кольца. Положительным свойством выбора такого соответствия является тот факт, что для любых двух вершин в кольце, расстояние между которыми является равным $l = 2^i$ для некоторого значения i, путь между соответствующими вершинами в гиперкубе содержит только две линии связи (за исключением случая $i = 0$, когда путь в гиперкубе имеет единичную длину).

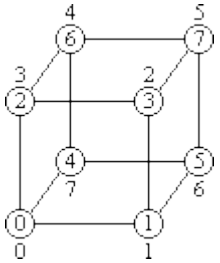

Рис. 3.1. Схема отображения гиперкуба на кольцо (в кружках приведены номера процессоров гиперкуба)

Представим величину сдвига q в виде двоичного кода. Количество ненулевых позиций кода определяет количество этапов в схеме реализации операции циклического сдвига. На каждом этапе выполняется операция сдвига с величиной шага, определяемой наиболее старшей ненулевой позицией значения q (например, при исходной величине сдвига $q = 5 = 101_2$, на первом этапе выполняется сдвиг с шагом 4, на втором этапе шаг сдвига равен 1). Выполнение каждого этапа (кроме сдвига с шагом 1) состоит в передаче данных по пути, включающему две линии связи. Как результат, верхняя оценка для длительности выполнения операции циклического сдвига определяется соотношением:

$t_{nд} = (t_н + mt_k)(2 \log p - 1)$

2. **Передача пакетов**. Использование пересылки пакетов может повысить эффективность выполнения операции циклического сдвига для топологии **гиперкуб**. Реализация всех необходимых коммуникационных действий в этом случае может быть обеспечена путем отправления каждым процессором всех пересылаемых данных непосредственно процессорам назначения. Использование метода покоординатной маршрутизации (см. п. 3.2) позволит избежать коллизий при использовании линий передачи данных (в каждый момент времени для каждого канала будет существовать не более одного готового для отправки сообщения). Длина наибольшего пути при такой рассылке данных определяется как $\log p - \gamma(q)$, где $\gamma(q)$ есть наибольшее целое значение j такое, что 2^j есть делитель величины сдвига q. Тогда длительность операции циклического сдвига может быть определена при помощи выражения

$t_{nд} = t_н + mt_k + t_c\ (\log p - \gamma(q))$

(при достаточно больших размерах сообщений временем передачи служебных данных можно пренебречь и время выполнения операции может быть определено как $t_{nд} = t_н + mt_k$).

3.4. Методы логического представления топологии коммуникационной среды

Как показало рассмотрение основных коммуникационных операций в п. 3.3, ряд алгоритмов передачи данных допускает более простое изложение при использовании вполне

определенных топологий сети межпроцессорных соединений. Кроме того, многие методы коммуникации могут быть получены при помощи того или иного логического представления исследуемой топологии. Как результат, важным моментом является при организации параллельных вычислений умение *логического представления разнообразных топологий* на основе конкретных (физических) межпроцессорных структур.

Способы логического представления (отображения) топологий характеризуются следующими тремя основными характеристиками:

- **уплотнение дуг** (*congestion*), выражаемое как максимальное количество дуг логической топологии, отображаемых в одну линию передачи физической топологии;
- **удлинение дуг** (*dilation*), определяемое как путь максимальной длины физической топологии, на который отображаемая дуга логической топологии;
- **увеличение вершин** (*expansion*), вычисляемое как отношение количества вершин в физической и логической топологиях.

Для рассматриваемых в рамках пособия топологий ограничимся изложением вопросов отображения топологий кольца и решетки на гиперкуб; предлагаемые ниже подходы для логического представления топологий характеризуются единичными показателями уплотнения и удлинения дуг.

Представление кольцевой топологии в виде гиперкуба

Установление соответствия между кольцевой топологией и гиперкубом может быть выполнено при помощи *двоичного рефлексивного кода Грея* $G(i, N)$ (*binary reflected Gray code*),

Код Грея для $N = 1$	Код Грея для $N = 2$	Код Грея для $N = 3$	Номера процессоров	
			гиперкуба	кольца
0	0 0	0 0 0	0	0
1	0 1	0 0 1	1	1
	1 1	0 1 1	3	2
	1 0	0 1 0	2	3
		1 1 0	6	4
		1 1 1	7	5
		1 0 1	5	6
		1 0 0	4	7

Рис. 3.2. Отображение кольцевой топологии на гиперкуб при помощи кода Грея

определяемого в соответствии с выражениями:

$G(0, 1) = 0$, $G(1, 1) = 1$,

$$G(i, s+1) = \begin{cases} G(i, s), & i < 2^s, \\ 2^s + G(2^{s+1} - 1 - i, s), & i \geq 2^s, \end{cases}$$

где i задает номер значения в коде Грея, а N есть длина этого кода. Для иллюстрации подхода на рис. 3.2 показывается отображение кольцевой топологии на гиперкуб для сети из $p = 8$ процессоров.

Важным свойством кода Грея является тот факт, что соседние значения $G(i, N)$ и $G(i + 1, N)$ имеют только одну различающуюся битовую позицию. Как результат, соседние вершины в кольцевой топологии отображаются на соседние процессоры в гиперкубе.

Отображение топологии решетки на гиперкуб

Отображение топологии решетки на гиперкуб может быть выполнено в рамках подхода, использованного для кольцевой структуры сети. Тогда для отображения решетки $2^r \times 2^s$ на гиперкуб размерности $N = r + s$ можно принять правило, что элементу решетки с координатами (i, j), будет соответствовать процессор гиперкуба с номером

$G(i, r) \,||\, G(j, s)$,

где операция || означает конкатенацию кодов Грея

3.5. Оценка трудоемкости операций передачи данных для кластерных систем

Для кластерных вычислительных систем (см. п. 1.3) одним из широко применяемых способов построения коммуникационной среды является использование концентраторов (*hub*) или переключателей (*switch*) для объединения процессорных узлов кластера в единую вычислительную сеть. В этих случаях топология сети кластера представляет собой *полный граф*, в котором, однако, имеются определенные ограничения на одновременность выполнения коммуникационных операций. Так, при использовании концентраторов передача данных в каждый текущий момент времени может выполняться только между двумя процессорными узлами; переключатели могут обеспечивать взаимодействие нескольких непересекающихся пар процессоров.

Другое часто применяемое решение при создании кластеров состоит в использовании *метода передачи пакетов* (реализуемого, как правило, на основе протокола TCP/IP) в качестве основного способа выполнения коммуникационных операций.

1. Выбирая для дальнейшего анализа кластеры данного распространенного типа (топология в виде полного графа, пакетный способ передачи сообщений), трудоемкость операции коммуникации между двумя процессорными узлами может быть оценена в соответствии с выражением (*модель А*)

$t_{пд}(m) = t_н + mt_k + t_c$,

оценка подобного вида следует из соотношений для метода передачи пакетов при единичной длине пути передачи данных, т.е. при $l = 1$. Отмечая возможность подобного подхода, вместе с этим можно заметить, что в рамках рассматриваемой модели время подготовки данных $t_н$ предполагается постоянным (не зависящим от объема передаваемых данных), время передачи служебных данных t_c не зависит от количества передаваемых пакетов и т.п. Эти предположения не в полной мере соответствуют действительности и временные оценки, получаемые в результате использования модели, могут не обладать необходимой точностью.

2. Учитывая все приведенные замечания, схема построения временных оценок может быть уточнена; в рамках новой расширенной модели трудоемкость передачи данных между двумя процессорами определяется в соответствии со следующими выражениями (*модель В*):

$$t_{пд} = \begin{cases} t_{нач_0} + m \cdot t_{нач_1} + (m + V_c) \cdot t_к, & n = 1 \\ t_{нач_0} + (V_{\max} - V_c) \cdot t_{нач_1} + (m + V_c \cdot n) \cdot t_к, & n > 1 \end{cases},$$

где $n = \lceil m / \ (V_{\max} - V_c) \rceil$ есть количество пакетов, на которое разбивается передаваемое сообщение, величина $V_{\max}$ определяет максимальный размер пакета, который может быть доставлен в сети (по умолчанию для операционной системы MS Windows в сети Fast Ethernet $V_{\max} = 1500$ байт), а V_c есть объем служебных данных в каждом из пересылаемых пакетов (для протокола TCP/IP, ОС Windows 2000 и сети Fast Ethernet $V_c = 78$ байт). Поясним также, что в приведенных соотношениях константа $t_{нач0}$ характеризует аппаратную составляющую латентности и зависит от параметров используемого сетевого оборудования, значение $t_{нач1}$ задает время подготовки одного байта данных для передачи по сети. Как результат, величина латентности

$$t_н = t_{нач0} + \nu \cdot t_{нач1}$$

увеличивается линейно в зависимости от объема передаваемых данных. При этом предполагается, что подготовка данных для передачи второго и всех последующих пакетов может быть совмещена с пересылкой по сети предшествующих пакетов и латентность, тем самым, не может превышать величины

$$t_н = t_{нач0} + (V_{\max} - V_c) \cdot t_{нач1}$$

Помимо латентности, в предлагаемых выражениях для оценки трудоемкости коммуникационной операции уточнено и правило вычисления времени передачи данных

$$(m + V_c \cdot n) \cdot t_к$$

что позволяет теперь учитывать эффект увеличения объема передаваемых данных при росте числа пересылаемых пакетов за счет добавления служебной информации (заголовков пакетов).

3. Завершая анализ проблемы построения теоретических оценок трудоемкости коммуникационных операций, следует отметить, что для практического применения перечисленных моделей необходимо выполнить оценку значений параметров используемых соотношений. В этом отношении полезным может оказаться использование и более простых способов вычисления временных затрат на передачу данных - среди известных схем подобного вида подход, в котором трудоемкость операции коммуникации между двумя процессорными узлами кластера оценивается в соответствии с выражением (*модель С*)

$$t_{пд}(m) = t_н + m / R,$$

где R есть пропускная способность сети передачи данных.

4. Для проверки адекватности рассмотренных моделей реальным процессам передачи данных приведем результаты выполненных экспериментов в сети многопроцессорного кластера Нижегородского университета (компьютеры IBM PC Pentium 4 1300 Мгц, 256 MB RAM, 10/100 Fast Etherrnet card). При проведении экспериментов для реализации коммуникационных операций использовалась библиотека MPI [1].

Часть экспериментов была выполнена для оценки параметров моделей:

- значение латентности $t_н$ для моделей A и C определялось как время передачи сообщения нулевой длины;
- величина пропускной способности R устанавливалась максимально наблюдаемой в ходе экспериментов скорости передачи данных, т.е.

$$R = \max_m (t_{пд}(m) / m),$$

- значения величин $t_{нач0}$ и $t_{нач1}$ оценивались при помощи линейной аппроксимации времен передачи сообщений размера от 0 до $V_{\max}$.

В ходе экспериментов осуществлялась передача данных между двумя процессорами кластера, размер передаваемых сообщений варьировался от 0 до 8 Мб. Для получения более точных оценок выполнение каждой операции осуществлялось многократно (более 100000 раз), после чего результаты временных замеров усреднялись. Для иллюстрации ниже приведен результат одного эксперимента, при проведении которого размер передаваемых сообщений изменялся от 0 до 1500 байт с шагом 4 байта.

Рис. 3.3. Зависимость экспериментального времени и времени, полученного по моделям A, B, C, от объема данных

В табл. 3.3 приводятся ряд числовых данных по погрешности рассмотренных моделей трудоемкости коммуникационных операций (величина погрешности дается в виде относительного отклонения от реального времени выполнения операции передачи данных).

Таблица 3.3. Погрешность моделей трудоемкости операций передачи данных (по результатам вычислительных экспериментов)

Объем сообщения в байтах	Время передачи данных в мкс	Погрешность теоретической оценки времени выполнения операции передачи данных		
		Модель A	**Модель B**	Модель C
32	172,0269	-16,36%	3,55%	-12,45%
64	172,2211	-17,83%	0,53%	-13,93%
128	173,1494	-20,39%	-5,15%	-16,50%
256	203,7902	-7,70%	0,09%	-4,40%
512	242,6845	0,46%	-1,63%	3,23%
1024	334,4392	14,57%	0,50%	16,58%
2048	481,5397	22,33%	5,05%	23,73%
4096	770,6155	28,55%	18,13%	29,42%

В результате приведенных данных можно заключить, что использование новой предложенной модели (модели B) позволяет оценивать время выполняемых операция передачи данных с более высокой точностью.

4. Параллельные численные методы для решения типовых задач вычислительной математики

4.1. Вычисление частных сумм последовательности числовых значений

Рассмотрим для первоначального ознакомления со способами построения и анализа параллельных методов вычислений сравнительно простую *задачу нахождения частных сумм* последовательности числовых значений

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

,

где n есть количество суммируемых значений (данная задача известна также под названием *prefix sum problem* – см. п. 3.3).

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* – см. п. 3.3.)

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

Последовательный алгоритм суммирования

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

$$S = 0,$$
$$S = S + x_1, \ldots$$

Вычислительная схема данного алгоритма может быть представлена следующим образом (см. рис. 4.1):

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

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

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

есть множество дуг, определяющих информационные зависимости операций.

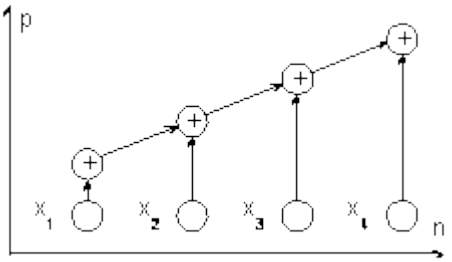

Рис. 4.1. Последовательная вычислительная схема алгоритма суммирования

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

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. 4.2):

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

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

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

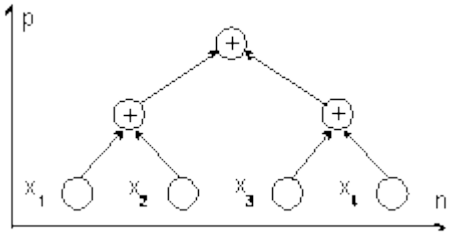

Рис. 4.2. Каскадная схема алгоритма суммирования

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

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

Как можно оценить, количество итераций каскадной схемы оказывается равным величине

$$k = \log_2 n,$$

а общее количество операций суммирования

$$L_{посл.} = n/2 + n/4 + \ldots + 1 = n - 1$$

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

$$L_{пар} = \log_2 n.$$

Как результат, можно оценить показатели ускорения и эффективности каскадной схемы алгоритма суммирования

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

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

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 паракомпьютера в теореме 2 (см. раздел 2). Однако при этом эффективность использования процессоров уменьшается при увеличении количества суммируемых значений

$$\lim E_p \to 0 \quad при \quad n \to \infty.$$

Модифицированная каскадная схема

Получение асимптотически ненулевой эффективности может быть обеспечено, например, при использовании модифицированной каскадной схемы [18]. В новом варианте каскадной схемы все проводимые вычисления подразделяется на два последовательно выполняемых этапа суммирования (см. рис. 4.3):

- 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.

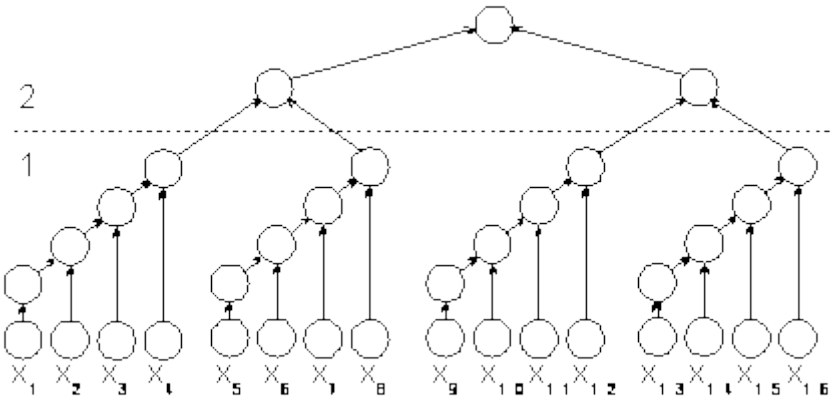

Рис. 4.3. Модифицированная каскадная схема суммирования

Для упрощения построения оценок можно предположить $n = 2^k = k^3$. 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

$$\log_2(n/\log_2 n) \le \log_2 n$$

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

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

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

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

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

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

$$E_p = (n-1)/2n \ge 0.25, \ \lim E_p \to 0.5 \quad при \quad n \to \infty.$$

Можно отметить также, что данные значения показателей достигаются при количестве процессоров, определенном в теореме 5 (см. раздел 2).

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 того же самого 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 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. 4.4) [18]:

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

$$S \leftarrow S + Q.$$

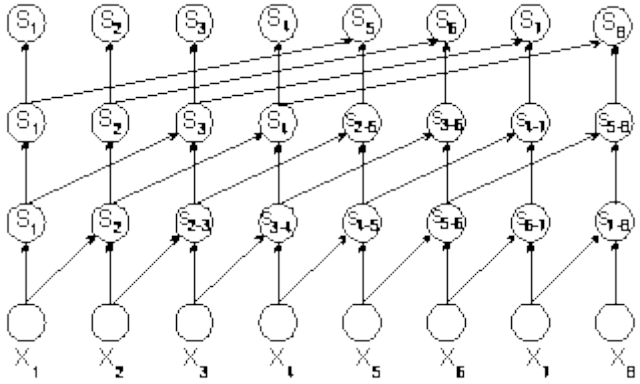

Рис. 4.4. Схема параллельного алгоритма вычисления всех частных сумм (величины S_{i-j} означают суммы значений от i до j элементов числовой последовательности)

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

$$L_{нар} = n \log_2 n$$

(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.

4.2. Умножение матрицы на вектор

Задача умножения матрицы на вектор определяется соотношениями

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \; 1 \le i \le n$$

.

Тем самым, получение результирующего вектора y предполагает повторения n однотипных операций по умножению строк матрицы A и вектора x. Получение каждой такой операции включает поэлементное умножение элементов строки матрицы и вектора x и последующее суммирование полученных произведений. Общее количество необходимых скалярных операций оценивается величиной

$$T_1 = 2n^2$$.

Как следует из выполняемых действий при умножении матрицы и вектора, параллельные способы решения задачи могут быть получены на основе параллельных алгоритмов суммирования (см. параграф 4.1). В данном разделе анализ способов распараллеливания будет дополнен рассмотрением вопросов организации параллельных вычислений в зависимости от количества доступных для использования процессоров. Кроме того, на примере задачи умножения матрицы на вектор будут обращено внимание на необходимость выбора наиболее подходящей топологии вычислительной системы (существующих коммуникационных каналов между процессорами) для снижения затрат для организации межпроцессорного взаимодействия.

Достижение максимально возможного быстродействия ($p = n^2$)

1. Выбор параллельного способа вычислений. Выполним анализ информационных зависимостей в алгоритме умножения матрицы на вектор для выбора возможных способов распараллеливания. Как можно заметить

- выполняемые при проведении вычислений операции умножения отдельных строк матрицы на вектор являются независимыми и могут быть выполнены параллельно;
- умножение каждой строки на вектор включает независимые операции поэлементного умножения и тоже могут быть выполнены параллельно;
- суммирование получаемых произведений в каждой операции умножения строки матрицы на вектор могут быть выполнены по одному из ранее рассмотренных вариантов алгоритма суммирования (последовательный алгоритм, обычная и модифицированная каскадные схемы).

Таким образом, максимально необходимое количество процессоров определяется величиной

$$p = n^2$$.

Использование такого количества процессоров может быть представлено следующим образом. Множество процессоров разбивается на групп

$$Q = (Q_1,\ldots,Q_n),$$

каждая из которых представляет набор процессоров для выполнения операции умножения отдельной строки матрицы на вектор. В начале вычислений на каждый процессор группы пересылаются элемент строки матрицы A и соответствующий элемент вектора x. Далее каждый процессор выполняет операцию умножения. Последующие затем вычисления выполняются по каскадной схеме суммирования. Для иллюстрации на рис. 4.5 приведена вычислительная схема для процессоров группы Q_i при размерности матрицы $n = 4$.

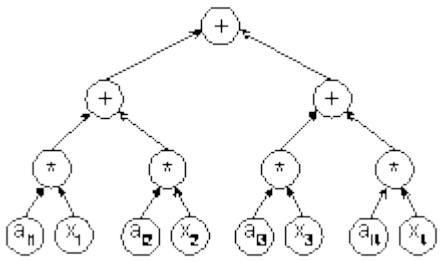

Рис. 4.5. Вычислительная схема операции умножения строки матрицы на вектор

2. Оценка показателей эффективности алгоритма. Время выполнения параллельного алгоритма при использовании $p = n^2$ процессоров определяется временем выполнения параллельной операции умножения и временем выполнения каскадной схемы

$$T_p = 1+\log_2 n.$$

Как результат, показатели эффективности алгоритма определяются следующими соотношениями:

$$p = n^2,\ S_p = 2n^2/(1+\log_2 n),$$

$$E_p = 2n^2/p(1+\log_2 n) = 2/(1+\log_2 n).$$

3. Выбор топологии вычислительной системы. Для рассматриваемой задачи умножения матрицы на вектор наиболее подходящими топологиями являются структуры, в которых обеспечивается быстрая передача данных (пути единичной длины) в каскадной схеме суммирования (см. рис. 4.5). Таковыми топологиями являются структура с полной системой связей (**полный граф**) и **гиперкуб**. Другие топологии приводят к возрастанию коммуникационного времени из-за удлинения маршрутов передачи данных. Так, при линейном упорядочивании процессоров с системой связей только с ближайшими соседями слева и справа (**линейка** или **кольцо**) для каскадной схемы длина пути передачи каждой получаемой частичной суммы на итерации i, $1 \le i \le \log_2 n$, является равной 2^{i-1}. Если принять, что передача данных по маршруту длины l в топологиях с линейной структурой требует выполнения l операций передачи данных, общее количество параллельных операций (суммарной длительности путей) передачи данных определяется величиной

$$L_{лд} = 1+2+\ldots+2^{\log_2(n/2)} = n-1$$

(без учета передач данных для начальной загрузки процессоров).

Применение вычислительной системы с топологией в виде прямоугольной **двумерной решетки** размера $n \times n$ приводит к простой и наглядной интерпретации выполняемых вычислений (структура сети соответствует структуре обрабатываемых данных). Для такой

топологии строки матрицы A наиболее целесообразно разместить по горизонталям решетки; в этом случае элементы вектора x должны быть разосланы по вертикалям вычислительной системы. Выполнение вычислений при таком размещении данных может осуществляться параллельно по строкам решетки; как результат, общее количество передач данных совпадает с результатами для линейки ($L_{кв} = n - 1$).

Коммуникационные действия, выполняемые при решении поставленной задачи, состоят в передаче данных между парами процессоров МВС. Подробный анализ длительности реализации таких операций проведен в п. 3.3.

4. Рекомендации по реализации параллельного алгоритма. При реализации параллельного алгоритма целесообразно выделить начальный этап по загрузке используемых процессоров исходными данными. Наиболее просто такая инициализация обеспечивается при топологии вычислительной системы с топологией в виде **полного графа** (загрузка обеспечивается при помощи одной параллельной операции пересылки данных). При организации множества процессоров в виде **гиперкуба** может оказаться полезной двухуровневое управление процессом начальной загрузки, при которой центральный управляющий процессор обеспечивает рассылку строк матрицы и вектора к управляющим процессорам процессорных групп Q_i, $1 \le i \le n$, которые, в свою очередь, рассылают элементы строк матрицы и вектора по исполнительным процессорам. Для топологий в виде **линейки** или **кольца** требуется n^2 последовательных операций передачи данных с последовательно убывающим объемом пересылаемых данных от $n(n+1)$ до 2 элементов.

Использование параллелизма среднего уровня ($n < p < n^2$)

1. Выбор параллельного способа вычислений. При уменьшении доступного количества используемых процессоров ($p < n^2$) обычная каскадная схема суммирования при выполнении операций умножения строк матрицы на вектор становится не применимой. Для простоты изложения материала положим $p = nk$ и воспользуется модифицированной каскадной схемой. Начальная нагрузка каждого процессора в этом случае увеличивается и процессор загружается (n/k) частями строк матрицы A и вектора x. Время выполнения операции умножения матрицы на вектор может быть оценено как величина

$$T_p = 2(n/k) + \log_2(k) = 2(n/(p/n)) + \log_2(p/n) =$$

$$= 2(n^2/p) + \log_2(p/n).$$

При использовании количества процессоров, необходимого для реализации модифицированной каскадной схемы, т.е. при $p = 2n(n/\log_2 n)$, данное выражение дает оценку времени исполнения $T_p \le 2T_\infty = 2\log_2 n$ (при $n \ge 4$).

При количестве процессоров $p = 2n$, когда время выполнения алгоритма оценивается как $T_p = n + 1$, может быть предложена новая схема параллельного выполнения вычислений, при которой для каждой итерации каскадного суммирования используются *неперекрывающиеся наборы процессоров*. При таком походе имеющегося количества процессоров оказывается достаточным для реализации только одной операции умножения строки матрицы A и вектора x. Кроме того, при выполнении очередной итерации каскадного суммирования процессоры, ответственные за исполнение всех предшествующих итераций, являются свободными. Однако этот недостаток предлагаемого подхода можно обратить в достоинство, задействовав простаивающие процессоры для обработки следующих

строк матрицы A. В результате может быть сформирована следующая схема *конвейерного* выполнения умножения матрицы и вектора:

- множество процессоров Q разбивается на непересекающиеся процессорные группы $Q = (Q_0, \ldots, Q_k)$, $k = \log_2 n$,

при этом группа Q_i, $1 \le i \le k$, состоит из $n/2^i$ процессоров и используется для выполнения i итерации каскадного алгоритма (группа Q_0 применяется для реализации поэлементного умножения); общее количество процессоров $p = 2n-1$;

- инициализация вычислений состоит в поэлементной загрузке процессоров группы Q_0 значениями 1 строки матрицы и вектора x; после начальной загрузки выполняется параллельная операция поэлементного умножения и последующей реализации обычной каскадной схемы суммирования;

- при выполнении вычислений каждый раз после завершения операции поэлементного умножения осуществляется загрузка процессоров группы Q_0 элементами очередной строки матрицы и инициируется процесс вычислений для вновь загруженных данных.

В результате применения описанного алгоритма множество процессоров Q реализует конвейер для выполнения операции умножения строки матрицы на вектор. На подобном конвейере одновременно могут находиться несколько отдельных строк матрицы на разных стадиях обработки. Так, например, после поэлементного умножения элементов первой строки и вектора x процессоры группы Q_1 будут выполнять первую итерацию каскадного алгоритма для первой строки матрицы, а процессоры группы Q_0 будут исполнять поэлементное умножение значений второй строки матрицы и т.д. Для иллюстрации на рис. 4.6 приведена ситуация процесса вычислений после 2 итераций конвейера при $n = 4$.

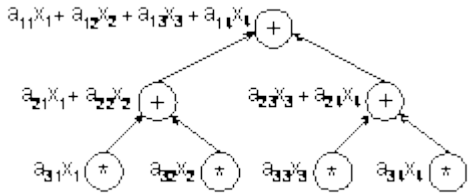

Рис. 4.6. Состояние конвейера для операции умножения строки матрицы на вектор после выполнения 2 итераций

2. Оценка показателей эффективности алгоритма. Умножение первой строки на вектор в соответствии с каскадной схемой будет завершено, как и обычно, после выполнения ($\log_2 n + 1$) параллельных операций. Для других же строк – в соответствии с конвейерной схемой организации вычислений - появление результатов умножения каждой очередной строки будет происходить после завершения каждой последующей итерации конвейера. Как результат, общее время выполнения операции умножения матрицы на вектор может быть выражено величиной

$$T_p = \log_2 n + 1 + n - 1 = \log_2 n + n.$$

Данная оценка является несколько большей, чем время выполнения параллельного алгоритма, описанного в предыдущем пункте ($T_p = n + 1$), однако вновь предлагаемый способ требует меньшего количества передаваемых данных (вектор x пересылается только однократно). Кроме того, использование конвейерной схемы приводит к более раннему появлению части результатов вычислений (что может быть полезным в ряде ситуаций обработки данных).

Как результат, показатели эффективности алгоритма определяются соотношениями следующего вида:

$p = 2n$, $S_p = 2n^2/(n+\log_2 n)$,

$E_p = 2n^2/p(n+\log_2 n) = n/(n+\log_2 n)$.

3. Выбор топологии вычислительной системы. Целесообразная топология вычислительной системы полностью определяется вычислительной схемой – это полное **бинарное дерево** высотой $\log_2 n+1$. Количество передач данных при такой топологии сети определяется общим количеством итераций, выполняемых конвейером, т.е.

$L_{под} = \log_2 n + n$.

Инициализация вычислений начинается с листьев дерева, результаты суммирования накапливаются в корневом процессоре.

Анализ трудоемкости выполняемых коммуникационных действий для вычислительных систем с другими топологиями межпроцессорных связей предполагается осуществить в качестве самостоятельного задания (см. также п. 3.4).

Организация параллельных вычислений при $p = n$

1. Выбор параллельного способа вычислений. При использовании n процессоров для умножения матрицы A на вектор x может быть использован ранее уже рассмотренный в пособии параллельный алгоритм построчного умножения, при котором строки матрицы распределяются по процессорам построчно и каждый процессор реализует операцию умножения какой-либо отдельной строки матрицы A на вектор x. Другой возможный способ организации параллельных вычислений может состоять в построении *конвейерной схемы для операции умножения строки матрицы на вектор* (*скалярного произведения векторов*) путем расположения всех имеющихся процессоров в виде линейной последовательности (**линейки**).

Подобная схема вычислений может быть определена следующим образом. Представим множество процессоров в виде линейной последовательности (см. рис. 4.7):

$Q = (q_1, \ldots, q_n)$;

каждый процессор q_j, $1 \le j \le n$, используется для умножения элементов j столбца матрицы и j элемента вектора x. Выполнение вычислений на каждом процессоре q_j, $1 \le j \le n$, состоит в следующем:

- запрашивается очередной элемент j столбца матрицы;
- выполняется умножение элементов a_{ij} и x_j;
- запрашивается результат вычислений S предшествующего процессора;
- выполняется сложение значений $S \leftarrow S + a_{ij}x_j$;
- полученный результат S пересылается следующему процессору.

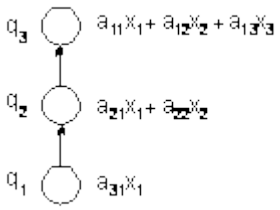

Рис. 4.7. Состояние линейного конвейера для операции умножения строки матрицы на вектор после выполнения двух итераций

При инициализации описанной схемы необходимо выполнить ряд дополнительных действий:

- при выполнении первой итерации каждый процессор дополнительно запрашивает элемент вектора x_j;
- для синхронизации вычислений (при выполнении очередной итерации схемы запрашивается результат вычисления предшествующего процессора) на этапе инициализации процессор q_j, $1 \le j \le n$, выполняет ($j-1$) цикл ожидания.

Кроме того, для однородности описанной схемы для первого процессора q_1, у которого нет предшествующего процессора, целесообразно ввести пустую операцию сложения ($S = 0, \quad S \leftarrow S + a_{ij} x_j$).

Для иллюстрации на рис. 4.7 показано состояние процесса вычислений после второй итерации конвейера при $n = 3$.

2. Оценка показателей эффективности алгоритма. Умножение первой строки на вектор в соответствии с описанной конвейерной схемой будет завершено после выполнения ($n+1$) параллельных операций. Результат умножения следующих строк будет происходить после завершения каждой очередной итерации конвейера (напомним, итерация каждого процессора включает выполнение операций умножения и сложения). Как результат, общее время выполнения операции умножения матрицы на вектор может быть выражено соотношением:

$$T_p = n + 1 + 2(n-1) = 3n - 1.$$

Данная оценка также является большей, чем минимально возможное время $T_p = 2n$ выполнения параллельного алгоритма при $p = n$. Полезность использования конвейерной вычислительной схемы состоит, как отмечалось в предыдущем пункте, в уменьшении количества передаваемых данных и в более раннем появлении части результатов вычислений.

Показатели эффективности данной вычислительной схемы определяются соотношениями:

$$p = n, \; S_p = 2n^2 / (3n-1),$$

$$E_p = 2n^2 / p(3n-1) = 2n/(3n-1).$$

3. Выбор топологии вычислительной системы. Необходимая топология вычислительной системы для выполнения описанного алгоритма однозначно определяется предлагаемой вычислительной схемой – это линейно упорядоченное множество процессоров (**линейка**).

Использование ограниченного набора процессоров ($p \le n$)

1. Выбор параллельного способа вычислений. При уменьшении количества процессоров до величины $p \le n$ параллельная вычислительная схема умножения матрицы на вектор может быть получена в результате адаптации алгоритма построчного умножения. В этом случае каскадная схема суммирования результатов поэлементного умножения вырождается и операция умножения строки матрицы на вектор полностью выполняется на единственном процессоре. Получаемая при таком подходе вычислительная схема может быть конкретизирована следующим образом:

- на каждый из имеющихся процессоров пересылается вектор x и $k = n / p$ строк матрицы;
- выполнение операции умножения строк матрица на вектор выполняется при помощи обычного последовательного алгоритма.

Следует отметить, что размер матрицы может оказаться не кратным количеству процессоров и тогда строки матрицы не могут быть разделены поровну между процессорами. В этих ситуациях можно отступить от требования равномерности загрузки процессоров и для получения более простой вычислительной схемы принять правило, что размещение данных на процессорах осуществляется только построчно (т.е. элементы одной строки матрицы не могут быть разделены между несколькими процессорами). Неодинаковое количество строк приводит к разной вычислительной нагрузке процессоров; тем самым, завершение вычислений (общая длительность решения задачи) будет определяться временем работы наиболее загруженного процессора (при этом часть от этого общего времени отдельные процессоры могут простаивать из-за исчерпания своей доли вычислений). Неравномерность загрузки процессоров снижает эффективность использования МВС и, как результат рассмотрения данного примера можно заключить, что *проблема балансировки* относится к числу важнейших задач параллельного программирования.

2. Оценка показателей эффективности алгоритма. Время выполнения параллельного алгоритма определяется оценкой

$$T_p = 2\lceil n / p \rceil n,$$

где величина $\lceil n / p \rceil$ есть наибольшее количество строк, загружаемых на один процессор. С учетом данной оценки показатели эффективности предлагаемой вычислительной схемы имеют вид:

$$S_p = 2n^2 / 2\lceil n / p \rceil n = n / \lceil n / p \rceil, \quad E_p = n / p\lceil n / p \rceil.$$

При кратности размера матрицы и количества процессоров показатели ускорения и эффективности алгоритма приводятся к виду:

$$S_p = p, \quad E_p = 1$$

и принимают, тем самым, максимально возможные значения.

3. Выбор топологии вычислительной системы. В соответствии с характером выполняемых межпроцессорных взаимодействий в предложенной вычислительной схеме в качестве возможной топологии МВС может служить организация процессоров в виде **звезды** (см. рис. 1.1). Управляющий процессор подобной топологии может использоваться для загрузки вычислительных процессоров исходными данными и для приема результатов выполненных вычислений.

4.3. Матричное умножение

Задача умножения матрицы на матрицу определяется соотношениями

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, 1 \le i, j \le n$$

.

(для простоты изложения материала будем предполагать, что перемножаемые матрицы A и B являются квадратными и имеют порядок $n \times n$).

Анализ возможных способов параллельного выполнения данной задачи может быть проведен по аналогии с рассмотрением задачи умножения матрицы на вектор. Оставив подобный анализ для самостоятельного изучения, покажем на примере задачи матричного умножения использование нескольких общих подходов, позволяющих формировать параллельные способы решения сложных задач.

Макрооперационный анализ алгоритмов решения задач

Задача матричного умножения требует для своего решения выполнение большого количества операций (n^3 скалярных умножений и сложений). Информационный граф алгоритма при большом размере матриц становится достаточно объемным и, как результат, непосредственный анализ этого графа затруднен. После выявления информационной независимости выполняемых вычислений могут быть предложены многочисленные способы распараллеливания алгоритма.

С другой стороны, алгоритм выполнения матричного умножения может быть рассмотрен как процесс решения n независимых подзадач умножения матрицы A на столбцы матрицы B . Введение макроопераций, как можно заметить по рис. 4.8, приводит к более компактному представлению информационного графа алгоритма, значительно упрощает проблему выбора эффективных способов распараллеливания вычислений, позволяет использовать типовые параллельные методы выполнения макроопераций в качестве конструктивных элементов при разработке параллельных способов решения сложных вычислительных задач.

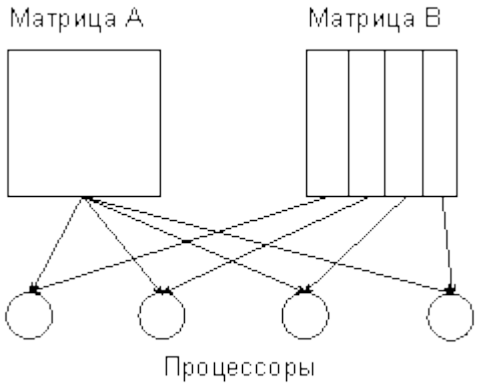

Рис. 4.8. Вычислительная схема матричного умножения при использовании макроопераций умножения матрицы A на столбец матрицы B

Важно отметить, что процесс введения макроопераций может осуществляться поэтапно с последовательно возрастающим уровнем детализации используемых операций. Так, для задачи матричного умножения после построения графа вычислений на основе макроопераций умножения матрицы на вектор может быть выполнено рассмотрение каждой макрооперации как последовательности независимых операций скалярного произведения векторов и т.п. Подобная *иерархическая декомпозиционная методика* построения параллельных методов решения сложных задач является одной из основных в параллельном программировании и широко используется в практике.

Организация параллелизма на основе разделения данных

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.

Пусть количество процессоров составляет $p = k^2$, а количество строк и столбцов матрицы является кратным величине $k = \sqrt{p}$, т.е. $n = mk$. Представим исходные матрицы A, B и результирующую матрицу C в виде наборов прямоугольных блоков размера $m \times m$. Тогда операцию матричного умножения матриц A и B в блочном виде можно представить следующим образом:

$$\begin{pmatrix} A_{11}A_{12}\dots A_{1k} \\ \dots \\ A_{k1}A_{k2}\dots A_{kk} \end{pmatrix} \times \begin{pmatrix} B_{11}B_{12}\dots B_{1k} \\ \dots \\ B_{k1}B_{k2}\dots B_{kk} \end{pmatrix} = \begin{pmatrix} C_{11}C_{12}\dots C_{1k} \\ \dots \\ c_{k1}C_{k2}\dots C_{kk} \end{pmatrix},$$

где каждый блок C_{ij} матрицы C определяется в соответствии с выражением

$$C_{ij} = \sum_{l=1}^{k} A_{il}B_{lj}.$$

Информационный граф алгоритма умножения при подобном представлении матриц показан на рис. 4.9 (на рисунке представлен фрагмент графа для вычисления только одного блока матрицы C). При анализе этого графа можно обратить внимание на взаимную независимость вычислений блоков C_{ij} матрицы C. Как результат, возможный подход для параллельного выполнения вычислений может состоять в выделении для расчетов, связанных с получением отдельных блоков C_{ij}, разных процессоров. Применение подобного подхода позволяет получить многие *эффективные параллельные методы умножения блочно-представленных матриц*; один из алгоритмов данного класса рассматривается ниже.

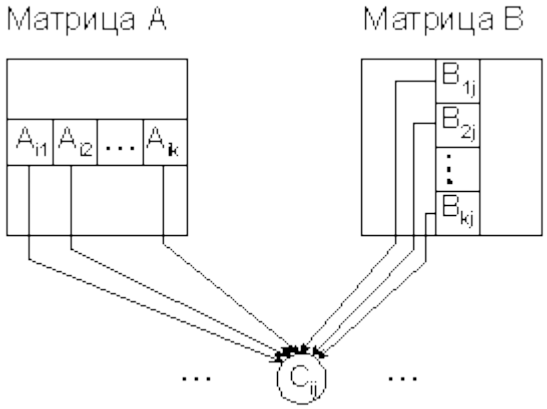

Рис. 4.9. Информационный граф матричного умножения при блочном представлении матриц

Для организации параллельных вычислений предположим, что процессоры образуют логическую прямоугольную решетку размером $k \times k$ (обозначим через p_{ij} процессор, располагаемый на пересечении i строки и j столбца решетки). Основные положения параллельного метода, известного как *алгоритм Фокса* (*Fox*) [15], состоят в следующем:

- каждый из процессоров решетки отвечает за вычисление одного блока матрицы C;

- в ходе вычислений на каждом из процессоров p_{ij} располагается четыре матричных блока:
  - блок C_{ij} матрицы C, вычисляемый процессором;
  - блок A_{ij} матрицы A, размещенный в процессоре перед началом вычислений;
  - блоки A'_{ij}, B'_{ij} матриц A и B, получаемые процессором в ходе выполнения вычислений;

Выполнение параллельного метода включает:

- **этап инициализации**, на котором на каждый процессор p_{ij} передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех процессорах;
- **этап вычислений**, на каждой итерации l, $1 \le l \le k$, которого выполняется:
  - для каждой строки i, $1 \le i \le k$, процессорной решетки блок A_{ij} процессора p_{ij} пересылается на все процессоры той же строки i; индекс j, определяющий положение процессора p_{ij} в строке, вычисляется по соотношению
$$j = (i + l - 1) \bmod k + 1,$$
($\bmod$ есть операция получения остатка от целого деления);
  - полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждого процессора p_{ij} перемножаются и прибавляются к блоку C_{ij}
$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij};$$
  - блоки B'_{ij} каждого процессора p_{ij} пересылаются процессорам p_{ij}, являющимися соседями сверху в столбцах процессорной решетки (блоки процессоров из первой строки решетки пересылаются процессорам последней строки решетки).

Для пояснения приведенных правил параллельного метода на рис. 4.10 приведено состояние блоков на каждом процессоре в ходе выполнения итераций этапа вычислений (для процессорной решетки 2×2).

Приведенный параллельный метод матричного умножения приводит к равномерному распределению вычислительной нагрузки между процессорами

$$T_p = 2n^3 / p, \quad S_p = 2n^3 / [p(2n^3 / p)] = 1.$$

Вместе с тем, блочное представление матриц приводит к некоторому повышению объема пересылаемых между процессорами данных - на этапе инициализации и на каждой итерации этапа вычислений на каждый процессор передается 2 блока данных общим объемом $2m^2$. Учитывая выполняемое количество итераций метода, объем пересылаемых данных для каждого процессора составляет величину

$$V_{ij} = 2m^2 + 2m^2 k = (2n^2 / \sqrt{p})(1 + 1/\sqrt{p})$$

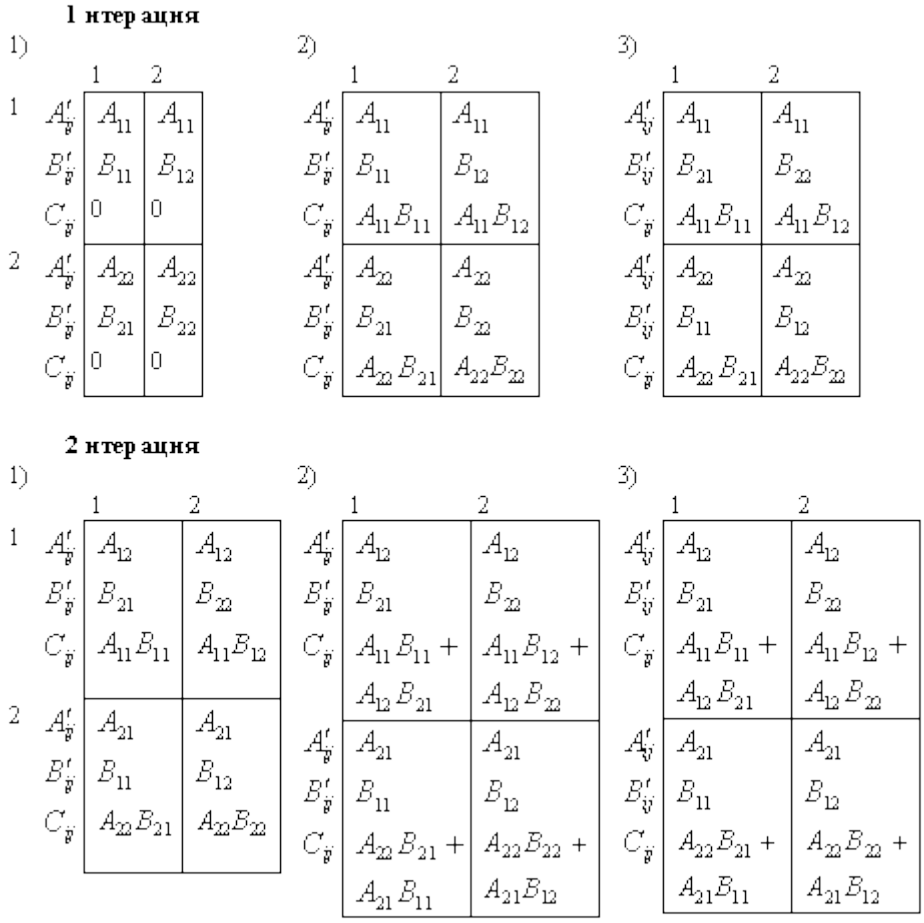

Рис. 4.10. Состояние блоков на каждом процессоре в ходе выполнения итераций этапа вычислений

Объем пересылаемых данных может быть снижен, например, при использовании строкового (для A) и столбцового (для B) разбиения матриц, при котором справедлива оценка

$$V'_{ij} = (2n^2 / \sqrt{p}) .$$

Данные оценки показывают, что различие объемов пересылаемых данных

$$V_{ij} / V'_{ij} = (1 + 1/\sqrt{p})(2n^2 / \sqrt{p}) / (2n^2 / \sqrt{p}) = (1 + 1/\sqrt{p})$$

является не столь существенным и данное различие уменьшается при увеличении числа используемых процессоров.

С другой стороны, использование блочного представления матриц приводит к ряду положительных моментов. Так, при данном способе организации вычислений пересылка данных оказывается распределенной по времени и это может позволить совместить процессы передачи и обработки данных; блочная структура может быть использована для создания высокоэффективных методов слабо заполненных (разреженных) матриц. И главное – данный метод является примером широко распространенного способа организации параллельных вычислений, состоящего в распределении между процессорами обрабатываемых данных с учетом близости их расположения в содержательных постановках решаемых задач. Подобная идея, часто называемая в литературе *геометрическим принципом распараллеливания*, широко используется при разработке параллельных методов решения сложных задач, поскольку во многих случаях может приводить к значительному снижению потоков пересылаемых данных за счет локализации на процессорах существенно информационно-

зависимых частей алгоритмов (в частности, подобный эффект может быть достигнут при численном решении дифференциальных уравнений в частных производных).

4.4. Сортировка

Сортировка является одной из типовых проблем обработки данных, и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = (a_1, a_2, \ldots, a_n)$$

в порядке монотонного возрастания или убывания

$$S \sim S' = ((a_1', a_2', \ldots, a_n') : a_1' \le a_2' \le \ldots \le a_n')$$

(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 [7]. 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 пересылаются 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.

Оставляя подробный анализ проблемы сортировки для специального рассмотрения, в пособии основное внимание уделяется изучению параллельных способов выполнения для ряда широко известных *методов внутренней сортировки*, когда все упорядочиваемые данные могут быть размещены полностью в оперативной памяти ЭВМ.

Параллельное обобщение базовой операции сортировки

1. При детальном 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

```
// операция "сравнить и переставить"
if ( a[i] > a[j] ) {
  temp = a[i];
```

```
  a[i] = a[j];
  a[j] = temp;
}
```

Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения собственно и проявляется различие алгоритмов сортировки. Так, например, в *пузырьковой сортировке* [7] осуществляется последовательное сравнение всех соседних элементов; в результате прохода по упорядочиваемому набору данных в последнем (верхнем) элементе оказывается максимальное значение ("всплывание пузырька"); далее для продолжения сортировки этот уже упорядоченный элемент отбрасывается и действия алгоритма повторяются

```
// пузырьковая сортировка
for ( i=1; i<n; i++ )
  for ( j=0; j<n-i; j++ )
    <сравнить и переставить элементы (a[j], a[j+1])>
}
```

2. Для параллельного обобщения выделенной базовой операции сортировки рассмотрим первоначально ситуацию, когда количество процессоров совпадает с числом сортируемых значений (т.е. $p = n$). Тогда сравнение значений a_i и a_j, располагаемых, например, на процессорах P_i и P_j, можно организовать следующим образом:

- выполнить взаимообмен имеющихся на процессорах P_i и P_j значений (с сохранением на этих процессорах исходных элементов);
- сравнить на каждом процессоре P_i и P_j получившиеся одинаковые пары значений (a_i, a_j); результаты сравнения используются для разделения данных между процессорами – на одном процессоре (например, P_i) остается меньший элемент, другой процессор (т.е. P_j) запоминает для дальней обработки большее значение пары

$a_i' = \min(a_i, a_j)$, $a_j' = \max(a_i, a_j)$.

3. Рассмотренное параллельное обобщение базовой операции сортировки может быть надлежащим образом адаптировано и для случая $p < n$, когда количество процессоров является меньшим числа упорядочиваемых значений. В данной ситуации каждый процессор будет содержать уже не единственное значение, а часть (*блок* размера n/p) сортируемого набора данных. Эти блоки обычно упорядочиваются в самом начале сортировки на каждом процессоре в отдельности при помощи какого-либо быстрого алгоритма (предварительная стадия параллельной сортировки). Далее, следуя схеме одноэлементного сравнения, взаимодействие пары процессоров P_i и P_j для совместного упорядочения содержимого блоков A_i и A_j может быть осуществлено следующим образом:

- выполнить взаимообмен блоков между процессорами P_i и P_j;
- объединить блоки A_i и A_j на каждом процессоре в один отсортированный блок двойного размера (при исходной упорядоченности блоков A_i и A_j процедура их объединения сводится к быстрой операции слияния упорядоченных наборов данных);
- разделить полученный двойной блок на две равные части и оставить одну из этих частей (например, с меньшими значениями данных) на процессоре P_i, а другую часть (с большими значениями соответственно) – на процессоре P_j

$$[A_i \cup A_j]_{спит} = A_i' \cup A_j' : \forall a_i' \in A_i', \forall a_j' \in A_j' \Rightarrow a_i' \le a_j'.$$

Рассмотренная процедура обычно именуется в литературе как операция "*сравнить и разделить*" (*compare-split*). Следует отметить, что сформированные в результате такой процедуры блоки на процессорах P_i и P_j совпадают по размеру с исходными блоками A_i и A_j и все значения, расположенные на процессоре P_i, являются меньшими значений на процессоре P_j.

Пузырьковая сортировка

Алгоритм пузырьковой сортировки [7], общая схема которого представлена в начале данного раздела, в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как метод *чет-нечетной перестановки* (*odd-even transposition*) [23]. 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. Т.е., на всех нечетных итерациях сравниваются пары

(a_1, a_2), (a_3, a_4),..., (a_{n-1}, a_n) (при четном n),

на четных итерациях обрабатываются элементы

(a_2, a_3), (a_4, a_5),..., (a_{n-2}, a_{n-1}).

После n-кратного повторения подобных итераций сортировки исходный набор данных оказывается упорядоченным.

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – сравнение пар значений на итерациях сортировки любого типа являются независимыми и могут быть выполнены параллельно. Для пояснений такого параллельного способа сортировки в табл. 4.1 приведен пример упорядочения данных при $n = 8$, $p = 4$ (т.е. блок значений на каждом процессоре содержит $n / p = 2$ элементов). В первом столбце таблицы приводится номер и тип итерации метода, перечисляются пары процессоров, для которых параллельно выполняются операция "сравнить и разделить"; взаимодействующие пары процессоров выделены в таблице двойной рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

Таблица 4.1. Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итерации	Процессоры			
	1	2	3	4
Исходны еданные	2 3	3 8	5 6	1 4
1 нечет (1,2),(3.4)	2 3	3 8	5 6	1 4
	2 3	3 8	1 4	5 6
2 чет (2,3)	2 3	3 8	1 4	5 6
	2 3	1 3	4 8	5 6
3 нечет (1,2),(3.4)	2 3	1 3	4 8	5 6
	1 2	3 3	4 5	6 8
4 чет (2,3)	1 2	3 3	4 5	6 8
	1 2	3 3	4 5	6 8

Следует отметить, что в приведенном примере последняя итерация сортировки является избыточной – упорядоченный набор данных был получен уже на третьей итерации алгоритма. В общем случае выполнение параллельного метода может быть прекращено, если в течение каких-либо двух последовательных итераций сортировки состояние упорядочиваемого набора данных не было изменено. Как результат, общее количество итераций может быть сокращено и для фиксации таких моментов необходимо введение некоторого управляющего процессора, который определял бы состояние системы после выполнения каждой итерации сортировки. Однако трудоемкость такой коммуникационной операции (сборка на одном процессоре сообщений от всех процессоров) может оказаться столь значительной, что весь эффект от возможного сокращения итераций сортировки будет поглощен затратами на реализацию операций межпроцессорной передачи данных.

Оценим трудоемкость рассмотренного параллельного метода. Длительность операций передач данных между процессорами полностью определяется физической топологией вычислительной сети. Если логически-соседние процессоры, участвующие в выполнении операций "сравнить и разделить", являются близкими фактически (например, для линейки или кольца эти процессоры имеют прямые линии связи), общая коммуникационная сложность алгоритма пропорциональна количеству упорядочиваемых данных, т.е. n. Вычислительная трудоемкость алгоритма определяется выражением

$$T_p = (n/p)\log(n/p) + 2n,$$

где первая часть соотношения учитывает сложность первоначальной сортировки блоков, а вторая величина задает общее количество операций для слияния блоков в ходе исполнения операций "сравнить и разделить" (слияние двух блоков требует $2(n/p)$ операций, всего выполняется p итераций сортировки). С учетом данной оценки показатели эффективности параллельного алгоритма имеют вид:

$$S_p = \frac{n\log n}{(n/p)\log(n/p) + 2n}, \quad E_p = \frac{n\log n}{p[(n/p)\log(n/p) + 2n]}$$

(в приведенных соотношениях для величины T_1 использовалась оценка трудоемкости для наиболее эффективных последовательных алгоритмов сортировки). Анализ выражений показывает, что если количество процессоров совпадает с числом сортируемых данных (т.е. $p = n$), эффективность использования процессоров падает с ростом n; получение асимптотически ненулевого значения показателя E_p может быть обеспечено при количестве процессоров, пропорциональных величине $\log n$.

Сортировка Шелла

Детальное описание *алгоритма Шелла* может быть получено, например, в [7]; здесь же отметим только, что общая идея метода 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).

Для алгоритма Шелла может быть предложен параллельный аналог метода, если топология коммуникационный сети имеет структуру N-мерного гиперкуба (т.е. количество процессоров равно $p = 2^N$). Выполнение сортировки в таком случае может быть разделено на два последовательных этапа. На первом этапе (N итераций) осуществляется взаимодействие процессоров, являющихся соседними в структуре гиперкуба (но эти процессоры могут оказаться далекими при линейной нумерации; для установления соответствия двух систем нумерации процессоров может быть использован, как и ранее, код Грея). 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 p. Трудоемкость параллельного варианта алгоритма Шелла определяется выражением:

$$T_p = (n/p)\log(n/p) + (2n/p)\log p + L(2n/p),$$

где вторая и третья части соотношения фиксируют вычислительную сложность первого и второго этапов сортировки соответственно. Как можно заметить, эффективность данного параллельного способа сортировки оказывается лучше показателей обычного алгоритма чет-нечетной перестановки при $L < p$.

Быстрая сортировка

При кратком изложении *алгоритм быстрой сортировки* (полное описание метода содержится, например, в [23]) 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 существует блок, все значения которого будут меньше 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 последовательно для обоих сформированных блоков и т.д. После выполнения $\log n$ итераций исходный массив данных оказывается упорядоченным.

Эффективность быстрой сортировки в значительной степени определяется успешностью 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 ($T_1 \sim n\log n$). В среднем случае количество операций, выполняемых алгоритмом быстрой сортировки, определяется выражением [7]:

$$T_1 = 1.4n\log n.$$

Параллельное обобщение алгоритма быстрой сортировки наиболее простым способом может быть получено для вычислительной системы с топологией в виде N-мерного гиперкуба (т.е. $p = 2^N$). Пусть, как и ранее, исходный набор данных распределен между процессорами

блоками одинакового размера n/p; результирующее расположение блоков должно соответствовать нумерации процессоров гиперкуба. Возможный способ выполнения первой итерации параллельного метода при таких условиях может состоять в следующем:

- выбрать каким-либо образом ведущий элемент и разослать его по всем процессорам системы;
- разделить на каждом процессоре имеющийся блок данных на две части с использованием полученного ведущего элемента;
- образовать пары процессоров, для которых битовое представление номеров отличается только в позиции N, и осуществить взаимообмен данными между этими процессорами; в результате таких пересылок данных на процессорах, для которых в битовом представлении номера бит позиции N равен 0, должны оказаться части блоков со значениями, меньшими ведущего элемента; процессоры с номерами, в которых бит N 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 на процессорах, в битовом представлении номеров которых бит N равен 0. Таких процессоров всего $p/2$ и, таким образом, исходный N-мерный гиперкуб также оказывается разделенным на два гиперкуба размерности $N-1$. К этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После N-кратного повторения подобных итераций для завершения сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре вычислительной системы. Для пояснения на рис. 4.11 представлен пример упорядочивания данных при $n=16$, $p=4$ (т.е. блок каждого процессора содержит 4 элемента). На этом рисунке процессоры изображены в виде прямоугольников, внутри которых показано содержимое упорядочиваемых блоков данных; значения блоков приводятся в начале и при завершении каждой итерации сортировки. Взаимодействующие пары процессоров соединены двунаправленными стрелками. Для разделения данных выбирались наилучшие значения ведущих элементов: на первой итерации для всех процессоров использовалось значение 0, на второй итерации для пары процессоров 0, 1 ведущий элемент равен 4, для пары процессоров 2, 3 это значение было принято равным –5.

Эффективность параллельного метода быстрой сортировки, как и в последовательном варианте, во многом зависит от успешности выбора значений ведущих элементов. Определение общего правила для выбора этих значений не представляется возможным; сложность такого выбора может быть снижена, если выполнить упорядочение локальных блоков процессоров перед началом сортировки и обеспечить однородное распределение сортируемых данных между процессорами вычислительной системы.

Длительность выполняемых операций передачи данных определяется операцией рассылки ведущего элемента на каждой итерации сортировки - общее количество межпроцессорных обменов для этой операции на N-мерном гиперкубе может быть ограничено оценкой

$$\sum_{i=1}^{N} i = N(N+1)/2 \sim \log^2 p,$$

и взаимообменом частей блоков между соседними парами процессоров – общее количество таких передач совпадает с количеством итераций сортировки, т.е. равно $\log p$, объем передаваемых данных не превышает удвоенного объема процессорного блока, т.е. ограничен величиной $2n/p$.

Вычислительная трудоемкость метода обуславливается сложностью локальной сортировки процессорных блоков, временем выбора ведущих элементов и сложностью разделения блоков, что в целом может быть выражено при помощи соотношения:

$$T_1 = (n/p)\log(n/p) + \log p + \log(n/p)\log p$$

(при построении данной оценки предполагалось, что для выбора значения ведущего элемента при упорядоченности процессорных блоков данных достаточно одной операции).

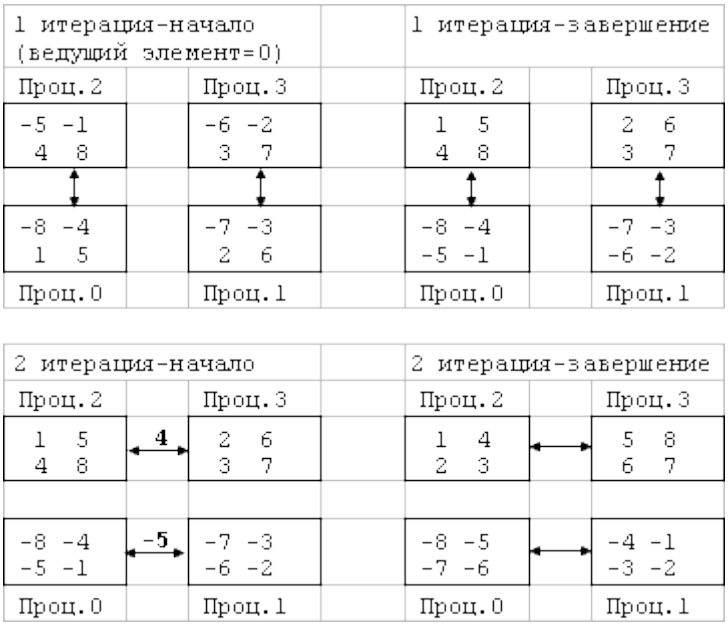

Рис. 4.11. Пример упорядочивания данных параллельным методом быстрой сортировки (без результатов локальной сортировки блоков)

4.5. 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, анализа и решения задач на графах посвящено достаточно много различных изданий, в качестве возможного руководства по данной тематике может быть рекомендована работа [8].

Пусть, как и ранее во 2 разделе пособия, G есть граф

$G = (V, R)$,

для которого набор вершин v_i, $1 \le i \le n$, задается множеством V, а список дуг графа

$r_j = (v_{s_j}, v_{t_j})$, $1 \le j \le k$,

определяется множеством R. В общем случае дугам графа могут приписываться некоторые числовые характеристики w_j, $1 \le j \le k$ (*взвешенный граф*).

Для описания графов известны различные способы задания. При малом количестве дуг в графе (т.е. $k << n^2$) целесообразно использовать для определения графов списки, перечисляющие имеющиеся в графах дуги. Представление достаточно плотных графов, для

которых почти все вершины соединены между собой дугами (т.е. $k \sim n^2$), может быть эффективно обеспечено при помощи *матрицы инцидентности*

$$A = (a_{ij}), \quad 1 \le i, j \le n,$$

ненулевые значения элементов которой соответствуют дугам графа

$$a_{ij} = \begin{cases} w(v_i, v_j), & если \quad (v_i, v_j) \in R, \\ 0, & если \quad i = j, \\ \infty, & иначе. \end{cases}$$

Использование матрицы инцидентности позволяет использовать также при реализации вычислительных процедур для графов матричные алгоритмы обработки данных.

Далее в пособии обсуждаются параллельные способы решения двух типовых задач на графах – нахождение минимально охватывающих деревьев и поиск кратчайших путей. Для представления графов используется способ задания при помощи матриц инцидентности.

Нахождение минимально 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*) [8]. Алгоритм начинает работу с произвольной вершины графа, выбираемого 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, $1 \le i \le n$, 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 ∞). В начале работы алгоритма выбирается корневая вершина МОД s и полагается

$$V_T = \{s\}, \quad d_s = 0.$$

Действия, выполняемые на каждой итерации алгоритма Прима, состоят в следующем:

- определяются значения величин d_i для всех вершин, еще не включенные в состав МОД;
- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T

$$t : d_t = \min d_i, i \notin V_T;$$

- включение выбранной вершины t в V_T.

После выполнения $n-1$ итераций 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 G

$$T_1 \sim n^2$$

.

Оценим возможности для параллельного выполнения рассмотренного алгоритма нахождения минимально 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 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. Соблюдение данного принципа приводит к тому, что при равномерной загрузке каждый процессор P_j, $1 \le j \le p$, будет содержать набор вершин

$$V_j = (v_{i_j+1}, v_{i_j+2}, \ldots, v_{i_j+k}), \quad i_j = k(j-1), \quad k = n/p,$$

соответствующий этому набору блок из k величин d_i, $1 \le i \le n$, и вертикальную полосу матрицы инцидентности графа G из k соседних столбцов, а также общую часть набора V_j и формируемого в процессе вычислений множества вершин V_T.

С учетом такого разделения данных итерация параллельного варианта алгоритма Прима состоит в следующем:

- определяются значения величин d_i для всех вершин, еще не включенные в состав МОД; данные вычисления выполняются независимо на каждом процессоре в отдельности; трудоемкость такой операции ограничивается сверху величиной n/p (на первой итерации алгоритма необходим перебор всех вершин, что требует вычислений порядка n^2/p);

- выбирается 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 процессоров (количество параллельных операций n/p), и выполнить сборку полученных значений (длительность такой операции передачи данных на гиперкубе, например, пропорциональна величине $\log p$);

- рассылка номера выбранной вершины для включения в охватывающее дерево всем процессорам (для гиперкуба сложность этой операции также определяется величиной $\log p$).

Получение МОД обеспечивается при выполнении n итераций алгоритма Прима; как результат, общая трудоемкость метода определяется соотношением

$$T_p = 2n^2 / p + 2n\log p .$$

С учетом данной оценки показатели эффективности параллельного алгоритма имеют вид:

$$S_p = \frac{n^2}{2n^2 / p + 2n\log p}, \quad E_p = \frac{n^2}{p[2n^2 / p + 2n\log p]} .$$

Анализ выражений показывает, что достижение асимптотически ненулевого значения показателя E_p становится возможным при количестве процессоров, пропорциональном величине $n / \log n$.

Поиск кратчайших путей

Задача поиска кратчайших путей на графе 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* [8], практически совпадает с методом Прима. Различие состоит лишь в интерпретации и в правиле оценки вспомогательных величин d_i, $1 \le i \le n$. В алгоритме Дейкстры эти величины означают суммарный вес пути от начальной вершины до всех остальных вершин графа. Как результат, после выбора очередной вершины t графа для включения в множество выбранных вершин V_T, значения величин d_i, $1 \le i \le n$, пересчитываются в соответствии с новым правилом:

$$\forall i \notin V_T \Rightarrow d_i = \min(d_i, d_t + w(t,i)) .$$

С учетом измененного правила пересчета величин d_i, $1 \le i \le n$, схема параллельного выполнения алгоритма Дейкстры может быть сформирована по аналогии с параллельным вариантом метода Прима. Конкретизация такой схемы и определение показателей эффективности метода для разных топологий вычислительной системы могут рассматриваться как темы самостоятельных заданий. Для сравнения в табл. 4.2 приводятся оценки числа процессоров, при которых достигается асимптотически ненулевые значения эффективности, для топологий кольца, решетки и гиперкуба.

Таблица 4.2. Показатели эффективности алгоритмов Прима и Дейкстры для разных топологий

Топология	Количество процессоров для асимптототической ненулевой эффективности	Трудоемкость при таком количестве процессоров
Кольцо	$\sqrt{n}$	$n^{1.5}$
Решетка	$n^{0.66}$	$n^{1.33}$
Гиперкуб	$n / \log n$	$n \log n$

5. Модели функционирования параллельных программ

Основой для построения моделей функционирования программ, реализующих параллельные методы решения задач, является понятие *процесса* как конструктивной единицы построения параллельной программы. Описание программы в виде набора

процессов, выполняемых параллельно на разных процессорах или на одном процессоре в режиме разделения времени, позволяет сконцентрироваться на рассмотрении проблем организации *взаимодействия процессов*, определить моменты и способы обеспечения с*инхронизации и взаимоисключения процессов*, изучить условия возникновения или доказать отсутствие *тупиков* в ходе выполнения программ (ситуаций, в которых все или часть процессов не могут быть продолжены при любых вариантах продолжения вычислений).

5.1. Концепция процесса

Понятие *процесса* является одним из основополагающих в теории и практике параллельного программирования. В [6,13] приводится ряд известных по литературе определений процесса. Разброс в трактовке данного понятия является достаточно широким, 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

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

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

$$t(p_n) = t_p = (\tau_1, \tau_2, \ldots, \tau_n),$$

где τ_j, $1 \le j \le n$, есть время начала выполнения команды τ_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

$$\forall i, \quad 1 \le i < n \quad \Rightarrow \quad \tau_{i+1} \ge \tau_i + 1.$$

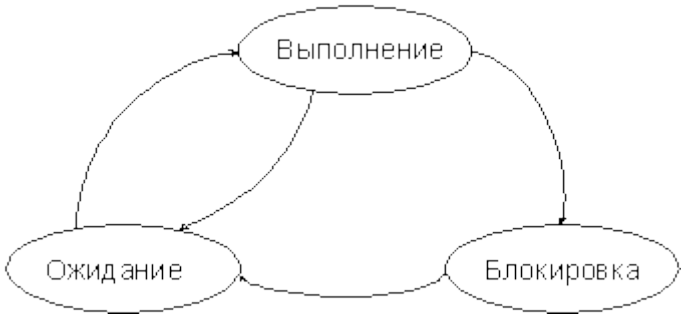

Рис. 5.1. Диаграмма переходов процесса из состояния в состояние

Равенство $\tau_{i+1} = \tau_i + 1$ 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 $\tau_{i+1} > \tau_i + 1$ 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). 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. 5.1.

5.2. 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.

5.3. Организация программ как системы процессов

Понятие процесса может быть использовано в качестве основного *конструктивного элемента* для построения параллельных программ в виде совокупности *взаимодействующих процессов*. 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.

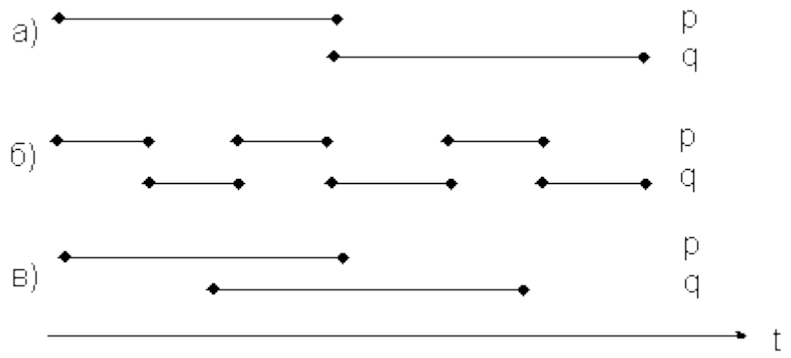

Рис. 5.2. Варианты взаиморасположения траекторий одновременно исполняемых процессов (отрезки линий изображают фрагменты командных последовательностей процессов)

Существование нескольких одновременно выполняемых процессов приводит к появлению дополнительных соотношений, которые должны выполняться для величин временных траекторий процессов. Возможные типовые варианты таких соотношений на примере двух процессов p и q состоят в следующем (см. рис. 5.2):

- выполнение процессов осуществляется строго последовательно, т.е. процесс q начинает свое выполнение только после полного завершения процесса p (*однопрограммный режим* работы ЭВМ – см. рис. 5.2а),

- выполнение процессов может осуществляться одновременно, но в каждый момент времени могут исполняться команды только какого либо одного процесса (*режим разделения времени* или *многопрограммный режим* работы ЭВМ – см. рис. 5.2б),

- *параллельное выполнение* процессов, когда одновременно могут выполняться команды нескольких процессов (данный режим исполнения процессов осуществим только при наличии в вычислительной системе нескольких процессоров - см. рис. 5.2в).

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

```
Процесс 1        Процесс 2
   N = N + 1        N = N + 1
   печать N         печать N
```

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

Время	Процесс 1	Процесс 2
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 процессов, если не предпринимать специальных мер, может зависеть от порядка исполнения команд.

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

$$p_n = (i_1, i_2, \cdots, i_n),\ q_m = (j_1, j_2, \cdots, j_m).$$

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

$$r_s = (l_1, l_2, \cdots, l_s),\ s = n + m.$$

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

$$x_s = (\chi_1, \chi_2, \cdots, \chi_s),$$

в котором следует положить $\chi_k = p_n$, если команда l_k получена из процесса p_n (иначе $\chi_k = q_m$). Порядок следования команд процессов в r_s должен соответствовать порядку расположения этих команд в исходных процессах

$$\forall u, v : (u < v), (\chi_u = \chi_v = p_n) \Rightarrow p_n(l_u) < p_n(l_v),$$

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

С учетом введенных обозначений, под программой, образованной из процессов 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*) процесс, получаемый путем параллельного объединения составляющих процессов

$$R_s = p_n \otimes q_m.$$

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

- моменты выполнения командных последовательностей разных процессов могут чередоваться по времени;
- между моментами исполнения команд разных процессов 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 одних и тех же исходных данных могут соответствовать разные командные последовательности вследствие разных вариантов чередования моментов работы разных процессов;
- доказательство правильности получаемых результатов должно проводиться для любых возможных временных соотношений для элементов временных траекторий процессов;
- для исключения зависимости результатов выполнения программы от порядка чередования команд разных процессов необходим анализ ситуаций взаимовлияния процессов и разработка методов для их исключения.

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

5.4. Взаимодействие и взаимоисключение процессов

Одной из причин зависимости результатов выполнения программ от порядка чередования команд может быть разделение одних и тех же данных между одновременно исполняемыми процессами (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 процессами необходимо иметь возможность:

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

Главным требованием к механизмам разделения ресурсов является гарантированное обеспечение *использования каждого разделяемого ресурса только одним процессом* от момента выделения ресурса этому процессу до момента освобождения ресурса. Данное требование в литературе обычно именуется *взаимоисключением процессов*; командные последовательности процессов, в ходе которых процесс использует ресурс, называется *критической секцией* процесса. С использованием последнего понятия условие взаимоисключения процессов может быть сформулировано как требование *нахождения в критических секциях по использованию одного и того же разделяемого ресурса не более чем одного процесса*.

Рассмотрим несколько вариантов программного решения проблемы взаимоисключения (для записи программ используется язык программирования C++). В каждом из вариантов будет предлагаться некоторый частный способ взаимоисключения процессов 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 процессов. Обсуждение способов взаимоисключения завершается рассмотрением *концепции семафоров Дейкстра*, которые могут быть использованы для общего решение проблемы взаимоисключения любого количества взаимодействующих процессов.

Попытка 1

```
intProcessNum = 1; // номер процесса для доступа к ресурсу
Process_1() {
  while (1) {
  // повторять, пока право доступа к ресурсу у процесса 2
  while
  ( ProcessNum == 2 );
  < Использование общего ресурса >
  // передача права доступа к ресурсу процессу 2
  ProcessNum = 2;
  }
}
Process_2()
  {
```

```
  while (1) {
  // повторять, пока право доступа к ресурсу у процесса 1
  while ( ProcessNum == 1 );
  < Использование общего ресурса >
  // передача права доступа к ресурсу процессу 1
  ProcessNum = 1;
  }
}
```

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

- ресурс используется процессами строго последовательно (по очереди) и, как результат, при разном темпе развития процессов общая скорость выполнения программы будет определяться наиболее медленным процессом;

- при завершении работы какого-либо процесса другой процесс не сможет воспользоваться ресурсом и может оказаться в постоянно блокированном состоянии.

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

Попытка 2

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

```
int ResourceProc1 = 0; // = 1 – ресурс занят
  процессом 1
int ResourceProc2 = 0; // = 1 – ресурс занят
  процессом 2
Process_1() {
  while (1) {
    // повторять, пока ресурс используется процессом 2
    while ( ResourceProc2 == 1 );
    ResourceProc1 = 1;
    < Использование общего ресурса >
    ResourceProc1 = 0;
  }
}
Process_2()
  {
  while (1) {
    // повторять, пока ресурс используется процессом 1
    while ( ResourceProc1 == 1 );
    ResourceProc2 = 1;
    < Использование общего ресурса >
    ResourceProc2 = 0;
  }
}
```

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

Следует отметить, что в отдельных случаях взаимоисключение процессов в данном примере может произойти и корректно - все определяется конкретными моментами переключения процессов. 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 процессов.

Попытка 3

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

```
int ResourceProc1 = 0; // = 1 – ресурс занят процессом 1
int ResourceProc2 = 0; // = 1 – ресурс занят процессом 2
Process_1() {
  while (1) {
    // установить, что процесс 1 пытается занять ресурс
    ResourceProc1 = 1;
    // повторять, пока ресурс занят процессом 2
    while ( ResourceProc2 == 1 );
    < Использование общего ресурса >
    ResourceProc1 = 0;
  }
}
Process_2()
  {
  while (1) {
    // установить, что процесс 2 пытается занять ресурс
    ResourceProc2 = 1;
    // повторять, пока ресурс используется процессом 1
    while ( ResourceProc1 == 1 );
    < Использование общего ресурса >
    ResourceProc2 = 0;
  }
}
```

Представленный вариант восстанавливает взаимоисключение, однако при этом возникает новая проблема – оба процесса 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 *тупика* (*дедлока* или *смертельного объятия*) и исключение тупиков является одной из наиболее важных задач в теории и практике параллельных вычислений. Более подробное рассмотрение темы будет выполнено далее в пп. 5.5 и 5.6; дополнительная информация по проблеме может быть получена в [6,13].

Попытка 4

Предлагаемый подход для устранения тупика состоит в организации временного снятия значения занятости управляющих переменных процессов в цикле ожидания ресурса.

```
int ResourceProc1 = 0; // =1 – ресурс занят
  процессом 1
int ResourceProc2 = 0; // =1 – ресурс занят
  процессом 2
Process_1() {
  while (1) {
    ResourceProc1 = 1; // процесс 1 пытается занять ресурс
    // повторять, пока ресурс занят процессом 2
    while ( ResourceProc2 == 1 ) {
      ResourceProc1 = 0; // снятие занятости ресурса
      < временная задержка >
      ResourceProc1 = 1;
```

```
    }
    < Использование общего ресурса >
    ResourceProc1 = 0;
  }
}
Process_2()
  {
  while (1) {
    ResourceProc2 = 1; // процесс 2 пытается занять ресурс
    // повторять, пока ресурс используется процессом 1
    while ( ResourceProc1 == 1 ) {
      ResourceProc2 = 0; // снятие занятости ресурса
      < временная задержка >
      ResourceProc2 = 1;
    }
    < Использование общего ресурса >
    ResourceProc2 = 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 процессов). Данная ситуация известна под наименованием *бесконечное откладывание* (*starvation*).

Алгоритм Деккера

В алгоритме Деккера предлагается объединение предложений вариантов 1 и 4 решения проблемы взаимоисключения.

```
int ProcessNum=1; // номер процесса для доступа
  к ресурсу
int ResourceProc1 = 0; // = 1 – ресурс занят
  процессом 1
int ResourceProc2 = 0; // = 1 – ресурс занят
  процессом 2
Process_1() {
  while (1) {
    ResourceProc1 = 1; // процесс 1 пытается занять ресурс
    /* цикл ожидания доступа к ресурсу */
    while ( ResourceProc2 == 1 ) {
      if ( ProcessNum == 2 ) {
        ResourceProc1 = 0;
        // повторять, пока ресурс занят процессом 2
        while ( ProcessNum == 2 );
        ResourceProc1 = 1;
      }
    }
    < Использование общего ресурса >
    ProcessNum  = 2;
    ResourceProc1 = 0;
  }
}
Process_2()
  {
  while (1) {
    ResourceProc2 = 1; // процесс 2 пытается занять ресурс
    /* цикл ожидания доступа к ресурсу */
    while ( ResourceProc1 == 1 ) {
```

```
        if ( ProcessNum == 1 ) {
          ResourceProc2 = 0;
          // повторять, пока ресурс используется процессом 1
          while ( ProcessNum == 1 );
          ResourceProc2 = 1;
        }
      }
      < Использование общего ресурса >
      ProcessNum = 1;
      ResourceProc2 = 0;
    }
  }
```

Алгоритм Деккера гарантирует корректное решение проблемы взаимоисключения для двух процессов. Управляющие переменные ResourceProc1, ResourceProc1 обеспечивают взаимоисключение, переменная ProcessNum исключает возможность бесконечного откладывания. Если оба процесса пытаются получить доступ к ресурсу, то процесс, номер которого указан в ProcessNum, продолжает проверку возможности доступа к ресурсу (внешний цикл ожидания ресурса). Другой же процесс 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 процессов (см. [16]), однако, такое обобщение приводит к заметному усложнению выполняемых действий. Кроме того, программное решение проблемы взаимоисключения процессов приводит к нерациональному использованию процессорного времени ЭВМ (процессу, ожидающему освобождения ресурса, постоянно требуется процессор для проверки возможности продолжения – *активное ожидание* (*busy wait*)).

Семафоры Дейкстры

Под *семафором S* обычно понимается [16] 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 процессов ожидают S >
  то < снять ожидание у одного из ожидающих процессов >
  иначе
  S = S + 1
```

Принципиальным в понимании семафоров является то, *что операции P(S) и V(S) предполагаются неделимыми*, 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 процессов. Так, например, проблема взаимоисключения при помощи семафоров может иметь следующее простое решение.

```
Semaphore
  Mutex=1; // семафор взаимоисключения процессов
Process_1() {
  while (1) {
    // проверить семафор и ждать, если ресурс занят
    P(Mutex);
    < Использование общего ресурса >
    // освободить один из ожидающих ресурса процессов
    // увеличить семафор, если нет ожидающих процессов
    V(Mutex);
  }
}
Process_2() {
  while (1) {
    // проверить семафор и ждать, если ресурс занят
    P(Mutex);
    < Использование общего ресурса >
    // освободить один из ожидающих ресурса процессов
    // увеличить семафор, если нет ожидающих процессов
    V(Mutex);
  }
}
```

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

5.5. Модель программы в виде дискретной системы

В самом общем виде *тупик* может быть определен [6] как ситуация, в которой один или несколько процессов ожидают какого-либо события, которое никогда не произойдет. 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 процесса, каждый из которых в монопольном режиме обрабатывает собственный файл данных. Ситуация тупика возникнет, например, если первому процессу для продолжения работы потребуются файл второго процесса и одновременно второму процессу окажется необходимым файл первого процесса (см. рис. 5.3).

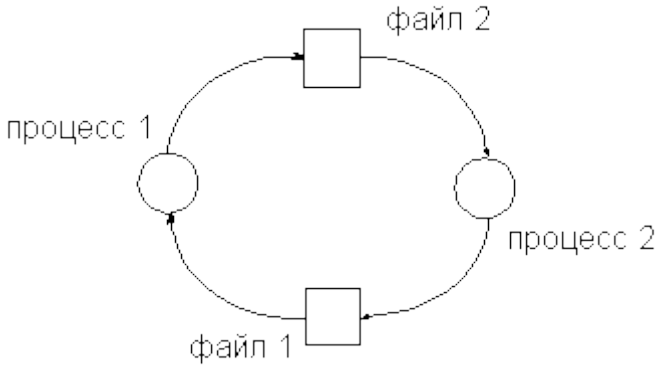

Рис. 5.3. Пример ситуации тупика

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 [6,13].

Могут быть выделены следующие **необходимые условия тупика** [13]:

- процессы требуют предоставления им права монопольного управления ресурсами, которые им выделяются (*условие взаимоисключения*);
- процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (условие *ожидания* ресурсов);
- ресурсы нельзя отобрать у процессов, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (условие *неперераспределяемости*);
- существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу 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 "процесс-ресурс", позволяющего обнаруживать ситуации кругового ожидания.

Определение состояния программы

Состояние программы может быть представлено в виде ориентированного графа (V,E) со следующей интерпретацией и условиями [13]:

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

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

и *ресурсы*

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

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 процесса 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 процессу 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.

$$\sum_i \left|(R_j, p_i)\right| \le k_j, \quad 1 \le j \le m;$$

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

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

Граф, построенный с соблюдением всех перечисленных правил, именуется в литературе как *граф "процесс-ресурс".* Для примера, на рис. 5.3 приведен граф программы, в которой ресурс 1 (файл 1) выделен процессу 1, который, в свою очередь, выдал запрос на ресурс 2 (файл 2). Процесс 2 владеет ресурсом 2 и нуждается для своего продолжения в ресурсе 1.

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

Запрос. Если программа находится в состоянии S и процесс 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 процессом p_i тогда и только тогда, когда p_i имеет запросы на выделение ресурсов и все такие запросы могут быть удовлетворены, т.е. если

$$\forall R_j : (p_i, R_j) \in E \quad \Rightarrow \quad (p_i, R_j) + \sum_i \left|(R_j, p_i)\right| \le k_j .$$

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.

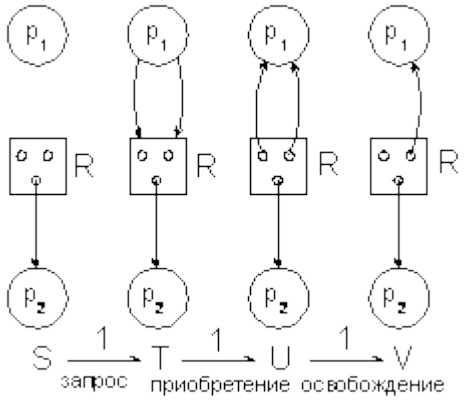

Рис. 5.4. Пример переходов программы из состояния в состояние

Освобождение. Процесс 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. 5.4. показаны состояния программы с одним ресурсом емкости 3 и двумя процессами после выполнения операций запроса, приобретения и освобождения ресурсов для первого процесса.

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

Описание возможных изменений 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 Σ есть множество состояний программы (S, T, U,…), а P представляет множество процессов $(p_1, p_2, \cdots, p_n)$. *Процесс* $p_i \in P$ есть частичная функция, отображающая состояния программы в непустые подмножества состояний

$$p_i : \Sigma \to (\Sigma) \;,$$

где (Σ) есть множество всех подмножеств Σ. Обозначим множество состояний, в которые может перейти программа при помощи процесса p_i (*область значений процесса* 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 процессе p_i (т.е. $T \in p_i(S)$) будем пояснять 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

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

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:

- *процесс* p_i *заблокирован* в состоянии S, если программа не может изменить свое состояние при помощи этого процесса, т.е. если $p_i(S) =$ Ø;

- *процесс* p_i находится в *тупике* в состоянии S, если этот процесс является заблокированным в любом состоянии T, достижимом из состояния S, т.е.

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

- *состояние* S называется *тупиковым*, если существует процесс p_i, находящийся в тупике в этом состоянии;

- *состояние* S есть *безопасное состояние*, если любое состояние T, достижимое из S, не является тупиковым.

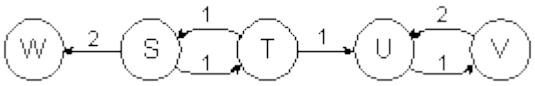

Рис. 5.5. Пример графа переходов программы

Для примера на рис. 5.5 приведен граф 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 [13].

Теорема. Граф "процесс-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 [13].

5.6. Модель программы в виде сети Петри

Другим возможным способом моделирования состояний и функционирования параллельной программы является использование математических моделей и методов исследования дискретных систем, разработанных в рамках теории сетей Петри [13]. При таком подходе в качестве модели программы может быть использована сеть Петри, представляемая размеченным ориентированным графом (V, E, M_0) (изложение материала осуществляется в соответствии с работой [13] за исключением приведения системы обозначений к виду, принятому в данном пособии):

1. Множество вершин сети V разделено на два взаимно пересекающихся подмножества *вершин-переходов* P и *вершин-мест* R

$$P = (p_1, p_2, \cdots, p_n), \quad R = (R_1, R_2, \cdots, R_m).$$

Вершины-места обычно изображаются кружками, вершины-переходы представляются в виде прямоугольников или линий-барьеров.

2. Граф сети является "двудольным" по отношению к подмножествам вершин P и R, т.е. каждое ребро $e \in E$ соединяет вершину P с вершиной R. Задание ребер графа может быть выполнено, например, при помощи функций инцидентности

$$F : R \times P \to \{0,1\}, \quad H : P \times R \to \{0,1\},$$

ненулевые значения которых задают множество ребер E (при $H(p_i, R_j) = 1$ сеть содержит ребро вида $e = (p_i, R_j)$, при $F(R_j, p_i) = 1$ сеть содержит ребро вида $e = (R_j, p_i)$).

3. Функция

$$M_0 : P \to \{0,1,2,\cdots\}$$

определяет собой *начальную разметку* сети, по которой для каждой вершины-места ставится в соответствие целое неотрицательное число (*разметка места*). При графическом изображении разметка сети показывается соответствующим числом точек (*фишек*) внутри кружков-мест.

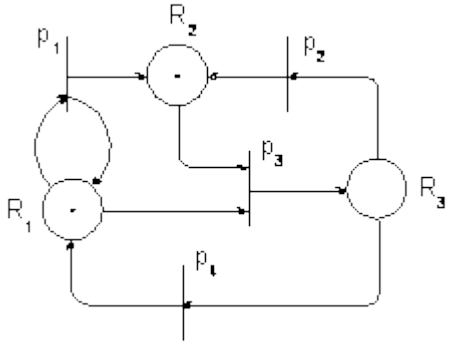

Рис. 5.6. Пример сети Петри

Для пояснения введенных понятий на рис. 5.6 показан пример сети Петри с тремя переходами и четырьмя вершинами мест.

При использовании сетей Петри для описания параллельных программ переходы обычно соответствуют действиям (процессам), а места – условиям (выделению или освобождению ресурсов). Разметка мест в этом случае интерпретируется как количество имеющихся (нераспределенных) единиц ресурса.

Сеть Петри может *функционировать* (изменять свое состояние), переходя от разметки к разметке. Обозначим через $F(R_j)$ множество переходов, к которым имеются ребра из вершины-места R_j

$$F(R_j) = \{p_i \in P : F(R_j, p_i) = 1\};$$

по аналогии, $H(R_j)$ есть множество переходов, из которых имеются ребра в вершину-место R_j

$$H(R_j) = \{p_i \in P : H(p_i, R_j) = 1\}$$

С учетом введенных обозначений правила функционирования сети состоят в следующем:

- Переход p_i может сработать при разметке M только при выполнении условия

$$\forall R_j \in R \Rightarrow M(R_j) - F(R_j, p_i) \geq 0.$$

Данное условие означает, что все входные места перехода p_i содержат хотя бы по одной фишке.

- В результате срабатывания перехода p_i разметка сети M сменяется разметкой M' по правилу:

$$\forall R_j \in R \Rightarrow M'(R_j) = M(R_j) - F(R_j, p_i) + H(p_i, R_j).$$

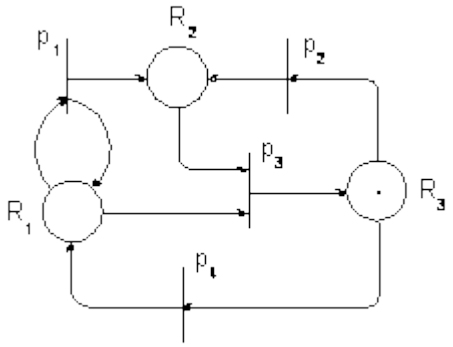

Рис. 5.7. Состояние сети после срабатывания перехода p_3

Другими словами, переход p_i изымает по одной фишке своего входного места и добавляет по одной в каждое свое выходное место. Будем говорить далее, что разметка M' следует за разметкой M, а M предшествует M', и обозначать этот факт

$$M \xrightarrow{p_i} M'.$$

Так, в сети на рис. 5.6 могут сработать переходы p_1 и p_3; состояние сети после срабатывания перехода p_3 показано на рис. 5.7.

Если одновременно может сработать несколько переходов и они не имеют общих входных мест, то их срабатывания могут рассматриваться как независимые действия, выполняемые последовательно или параллельно. Если несколько переходов могут сработать и имеют хотя бы одно общее входное место, то сработать может только один, любой из них.

При исследовании процессов функционирования сетей Петри широко используется следующий ряд дополнительных понятий и обозначений:

- разметка сети называется *тупиковой*, если при этой разметке ни один из переходов сети не может сработать;
- разметка M' *достижима* в сети от разметки M

$$M \to M',$$

если разметка M' может быть получена в результате некоторого количества срабатываний сети, начиная от разметки M;

- разметка M *достижима* в сети, если $M_0 \to M$; множество всех достижимых разметок обозначим через M;
- переход p_i *достижим* от разметки M, если существует достижимая от M разметка M', в которой переход p_i может сработать;
- переход p_i *достижим* в сети, если он достижим от M_0;
- переход p_i называется *живым*, если он достижим от любой разметки из M; сеть является *живой*, если все ее переходы живы;

- место R_j называется *ограниченным*, если существует такое число k, что $M(p) \le k$ для любой разметки из M; сеть является *ограниченной*, если все ее места ограничены.

В рамках теории сетей Петри разработаны методы, позволяющие для произвольной сети определить [26], является ли сеть ограниченной или живой, проверить достижимость любого перехода или разметки сети. Как результат, данные методы позволяют определить наличие тупиков в сети.

6. Учебно-практическая задача: 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, [5,11,19]).

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

$$\begin{cases} \frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = f(x,y), & (x,y) \in D, \\ u(x,y) = g(x,y), & (x,y) \in D^0, \end{cases}$$

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 [4, 10, 27].

Для простоты изложения материала в качестве области задания D функции $u(x,y)$ далее будет использоваться единичный квадрат

$$D = \{(x,y) \in D : 0 \le x,y \le 1\}.$$

6.1. Последовательные методы решения задачи Дирихле

Одним из наиболее распространенных подходов численного решения дифференциальных уравнений является *метод конечных разностей* (*метод сеток*) [5,11]. 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. 6.1)

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih,\ y_i = jh,\ 0 \le i,j \le N+1, \\ \qquad\qquad h = 1/(N+1), \end{cases}$$

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. 6.1) для вычисления значений производных, уравнение Пуассона может быть представлено 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}

$$u_{ij}=0.25(u_{i-1,j}+u_{i+1,j}+u_{i,j-1}+u_{i,j+1}-h^2f_{ij}).$$

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

$$u_{ij}^{k}=0.25(u_{i-1,j}^{k}+u_{i+1,j}^{k-1}+u_{i,j-1}^{k}+u_{i,j+1}^{k-1}-h^2f_{ij}),$$

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, [5,11]), здесь же отметим, что последовательность решений, получаемых методом сеток, равномерно сходится к решению задачи Дирихле, а погрешность решения имеет порядок h^2.

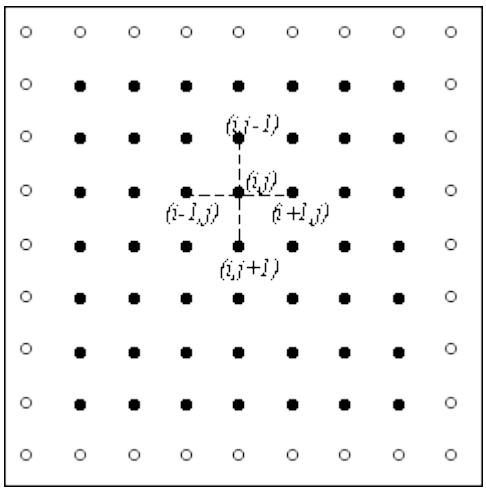

Рис. 6.1. Прямоугольная сетка в области *D* (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах - сверху вниз).

Рассмотренный алгоритм (метод Гаусса-Зейделя) на псевдокоде, приближенного к алгоритмическому языку C++, может быть представлен в виде:

```
// Алгоритм 6.1
 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 );
```

(напомним, что значения u_{ij} при индексах $i,j=0,N+1$ являются граничными, задаются при постановке задачи и не изменяются в ходе вычислений).

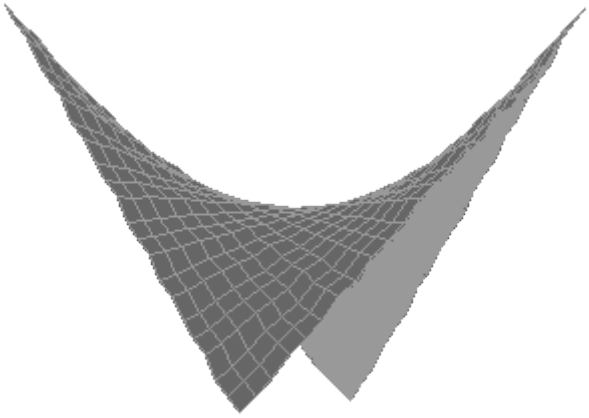

Рис. 6.2. Вид функции $u(x,y)$ в примере для задачи Дирихле

Для примера на рис. 6.2 приведен вид функции $u(x,y)$, полученной для задачи Дирихле при следующих граничный 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 210 при точности решения $eps=0.1$ и $N=100$ (в качестве начального приближения величин u_{ij} использовались значения, сгенерированные датчиком случайных чисел из диапазона [-100,100]).

6.2. 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.

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 имеющие 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 очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (*symmetric multiprocessors, SMP*).

Обычный подход 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 перечисленных 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, по которому 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* [17], 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 (*однопотоковых*) и параллельных (*многопотоковых*) участков программного кода (см. рис. 6.3). Подобный принцип организации параллелизма получил наименование "*вилочного*" (*fork-join*) или *пульсирующего параллелизма*. Более полная информация по технологии OpenMP может быть получена, например, в [30] или в информационных ресурсах сети Интернет; в пособии возможности OpenMP будут излагаться в объеме, необходимом для демонстрации возможных способов разработки параллельных программ для рассматриваемого учебного примера решения задачи Дирихле.

Проблема синхронизации параллельных вычислений

Первый вариант параллельного алгоритма для метод сеток может быть получен, если разрешить произвольный порядок пересчета значений u_{ij}. Программа для данного способа вычислений может быть представлена в следующем виде:

```
// Алгоритм 6.2
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,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 );
```

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).

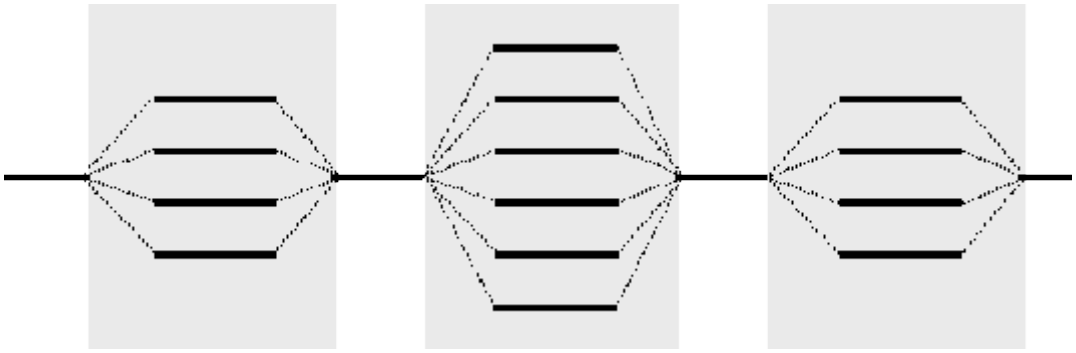

Рис. 6.3. Параллельные области, создаваемые директивами OpenMP

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; как отмечалось ранее, 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. 6.1 (здесь и далее для параллельных программ, разработанных с использованием технологии OpenMP, использовался четырехпроцессорный сервер кластера Нижегородского университета с процессорами Pentium III, 700 Mhz, 512 RAM).

Таблица 6.1. Результаты вычислительных экспериментов для параллельных вариантов алгоритма Гаусса-Зейделя (***p***=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 6.1)		Параллельный алгоритм 6.2			Параллельный алгоритм 6.3		
	k	***t***	***k***	***t***	***S***	***k***	***t***	***S***
100	210	0,06	210	1,97	0,03	210	0,03	2,03
200	273	0,34	273	11,22	0,03	273	0,14	2,43
300	305	0,88	305	29,09	0,03	305	0,36	2,43
400	318	3,78	318	54,20	0,07	318	0,64	5,90
500	343	6,00	343	85,84	0,07	343	1,06	5,64
600	336	8,81	336	126,38	0,07	336	1,50	5,88
700	344	12,11	344	178,30	0,07	344	2,42	5,00
800	343	16,41	343	234,70	0,07	343	8,08	2,03
900	358	20,61	358	295,03	0,07	358	11,03	1,87
1000	351	25,59	351	366,16	0,07	351	13,69	1,87
2000	367	106,75	367	1585,84	0,07	367	56,63	1,89
3000	370	243,00	370	3598,53	0,07	370	128,66	1,89

(***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 ускорение вычислений от использования нескольких процессоров окажется не столь существенным. 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 остальные

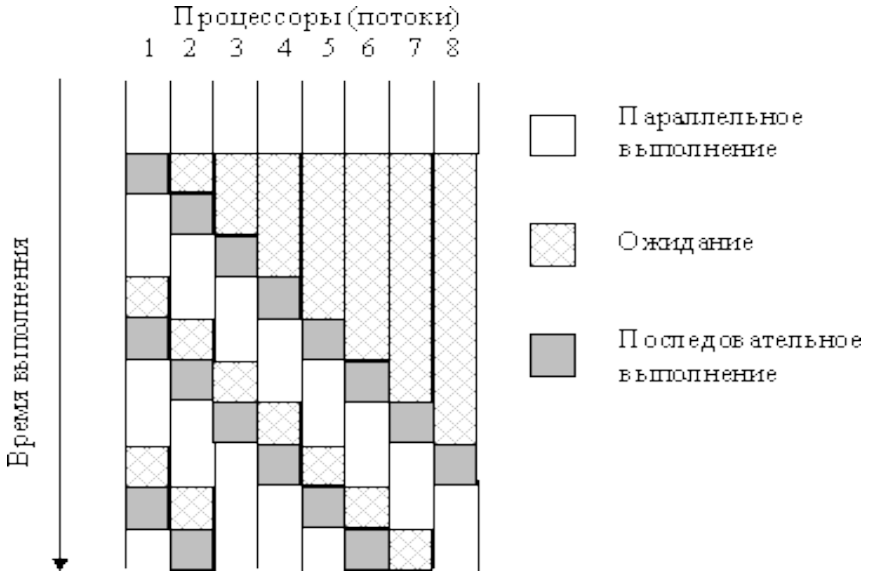

Рис. 6.4. Пример возможной схемы выполнения параллельных потоков при наличии синхронизации (взаимоисключения)

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. 6.4. Следует обратить внимание, что, несмотря на идеальное распределение вычислительной нагрузки между процессорами, для приведенного на рис. 6.4 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 **dm** с общей оценкой погрешности **dmax**.
Новый вариант программы решения задачи Дирихле имеет вид:

```
// Алгоритм 6.3
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,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 );
```

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. 6.1, показывают существенное изменение ситуации – ускорение в ряде экспериментов оказывается даже большим, чем используемое количество процессоров (такой эффект *сверхлинейного ускорения* достигается за счет наличия у каждого из процессоров вычислительного сервера своей быстрой кэш памяти). Следует также обратить внимание, что улучшение показателей параллельного алгоритма достигнуто при снижении максимально возможного параллелизма (для выполнения программы может использоваться не более N процессоров).

Возможность неоднозначности вычислений в параллельных программах

Последний рассмотренный вариант организации параллельных вычислений для метода сеток обеспечивает практически максимально возможное ускорение выполняемых расчетов – так, в экспериментах данное ускорение достигало величины 5.9 при использовании четырехпроцессорного 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. 6.5). 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 именуется *состязанием потоков* (*race condition*) и отражает важный принцип параллельного программирования – временная динамика выполнения параллельных потоков не должна учитываться при разработке параллельных алгоритмов и программ.

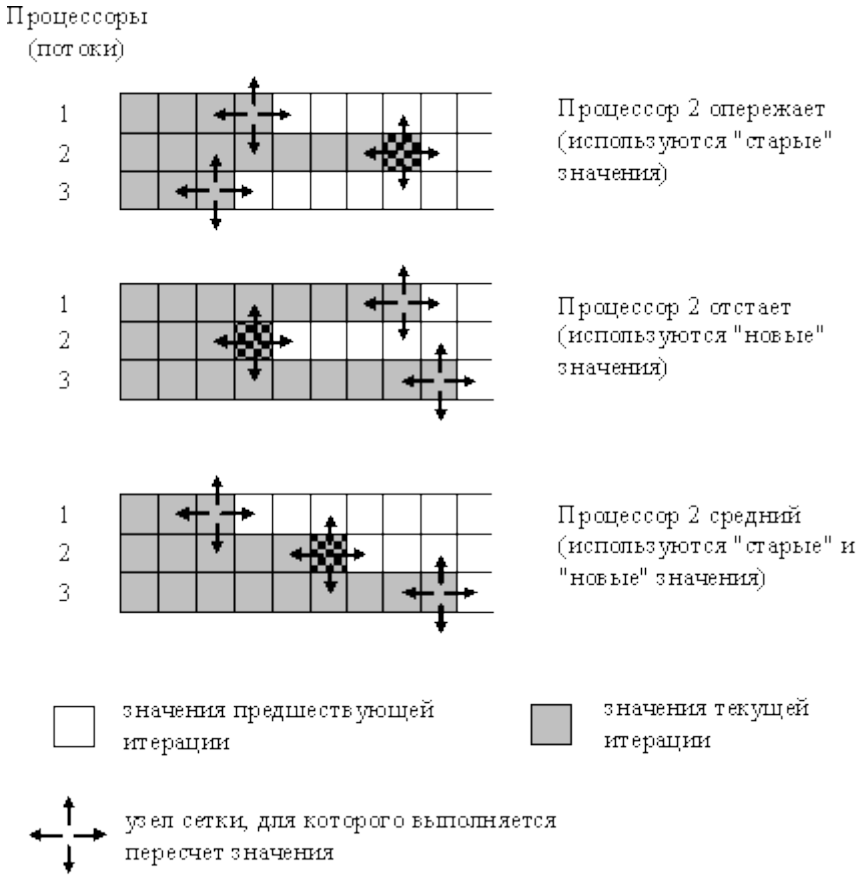

Рис. 6.5. Возможные различные варианты взаиморасположения параллельных потоков (состязание потоков)

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*).

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 строки 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. 6.6). В этом случае доступ к необходимым строкам не может быть обеспечен ни для одного потока – возникает неразрешимая ситуация, обычно именуемая *тупиком*. Как отмечалось в главе 5 пособия, необходимым условием тупика является наличие цикла в графе распределения и запросов ресурсов. 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 строки 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").

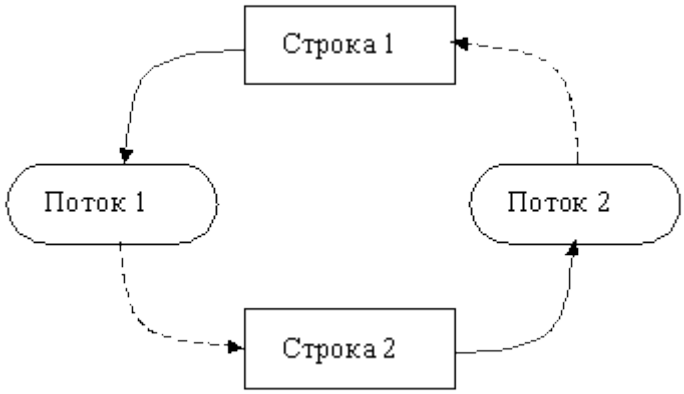

Рис. 6.6. Ситуация тупика при доступе к строкам сетки (поток 1 владеет строкой 1 и запрашивает строку 2, поток 2 владеет строкой 2 и запрашивает строку 1)

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 6.4
```

```
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,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 );
```

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 реализация сеточных алгоритмов обычно именуется *методом Гаусса-Якоби*. Этот метод гарантирует однозначность результаты 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. 6.2.

Таблица 6.2. Результаты вычислительных экспериментов для алгоритма Гаусса-Якоби (***p***=4)

Размер сетки	Последовательный метод Гаусса-Якоби (алгоритм 6.4)		Параллельный метод, разработанный по аналогии с алгоритмом 6.3		
	k	***t***	***k***	***t***	***S***
100	5257	1,39	5257	0,73	1,90
200	23067	23,84	23067	11,00	2,17
300	26961	226,23	26961	29,00	7,80
400	34377	562,94	34377	66,25	8,50
500	56941	1330,39	56941	191,95	6,93
600	114342	3815,36	114342	2247,95	1,70
700	64433	2927,88	64433	1699,19	1,72
800	87099	5467,64	87099	2751,73	1,99
900	286188	22759,36	286188	11776,09	1,93
1000	152657	14258,38	152657	7397,60	1,93
2000	337809	134140,64	337809	70312,45	1,91
3000	655210	247726,69	655210	129752,13	1,91

(***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. 6.7). 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.

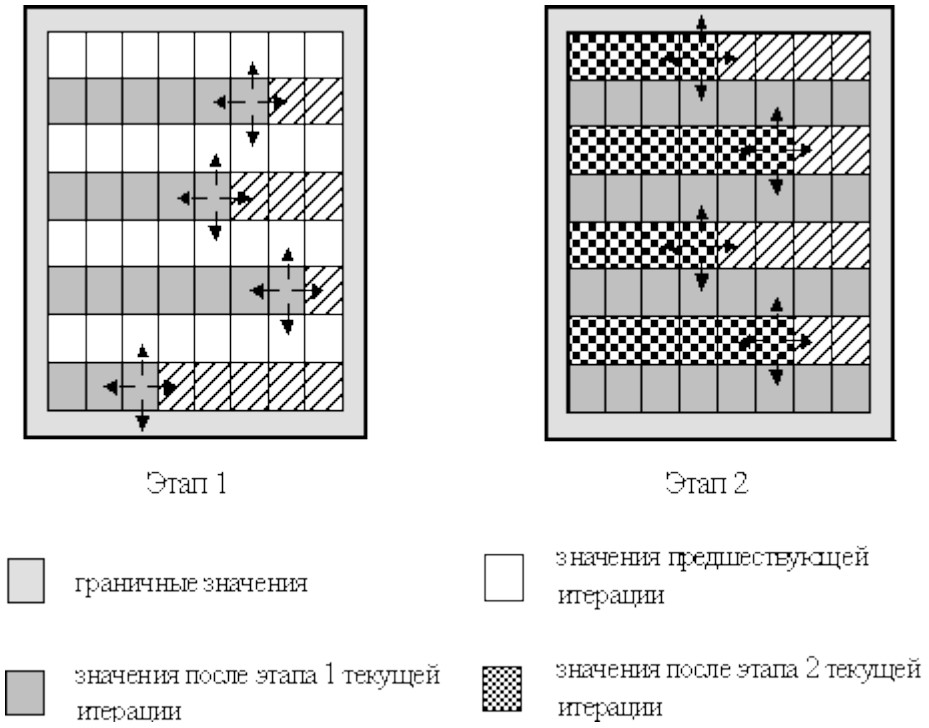

Рис. 6.7. Схема чередования обработки четных и нечетных строк

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. 6.8. Получаемая в результате вычислительная схема получила наименование *волны* или *фронта 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 6.3. Результаты экспериментов для параллельных вариантов алгоритма Гаусса-Зейделя с волновой схемой расчета (***p***=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 6.1)		Параллельный алгоритм 6.5			Параллельный алгоритм 6.6		
	k	***t***	***k***	***t***	***S***	***k***	***t***	***S***
100	210	0,06	210	0,30	0,21	210	0,16	0,40
200	273	0,34	273	0,86	0,40	273	0,59	0,58
300	305	0,88	305	1,63	0,54	305	1,53	0,57
400	318	3,78	318	2,50	1,51	318	2,36	1,60
500	343	6,00	343	3,53	1,70	343	4,03	1,49
600	336	8,81	336	5,20	1,69	336	5,34	1,65
700	344	12,11	344	8,13	1,49	344	10,00	1,21
800	343	16,41	343	12,08	1,36	343	12,64	1,30
900	358	20,61	358	14,98	1,38	358	15,59	1,32
1000	351	25,59	351	18,27	1,40	351	19,30	1,33
2000	367	106,75	367	69,08	1,55	367	65,72	1,62
3000	370	243,00	370	149,36	1,63	370	140,89	1,72

(***k*** – количество итераций, ***t*** – время в сек., ***S*** – ускорение)

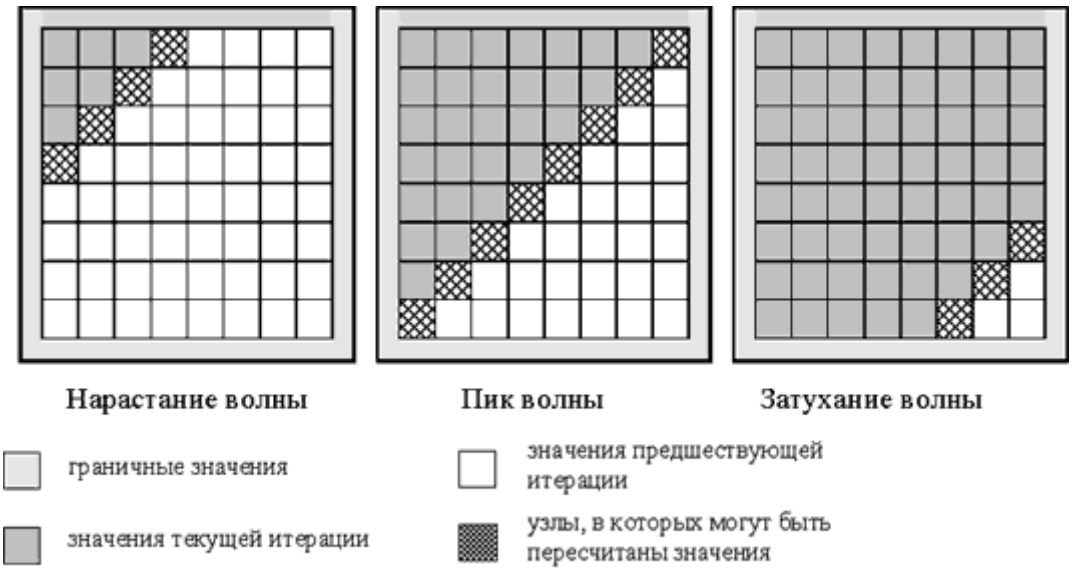

Рис. 6.8. Движение фронта волны вычислений

Возможная схема параллельного метода, основанного на эффекте волны вычислений, может быть представлена в следующей форме:

```
// Алгоритм 6.5
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
  // 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 );
```

При разработки алгоритма, реализующего волновую схему вычислений, оценку погрешности решения можно осуществлять для каждой строки в отдельности (массивы 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 среди 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. 6.3.

Следует обратить внимание на еще 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 [12]). 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 области расчетов - см. рис. 6.9.

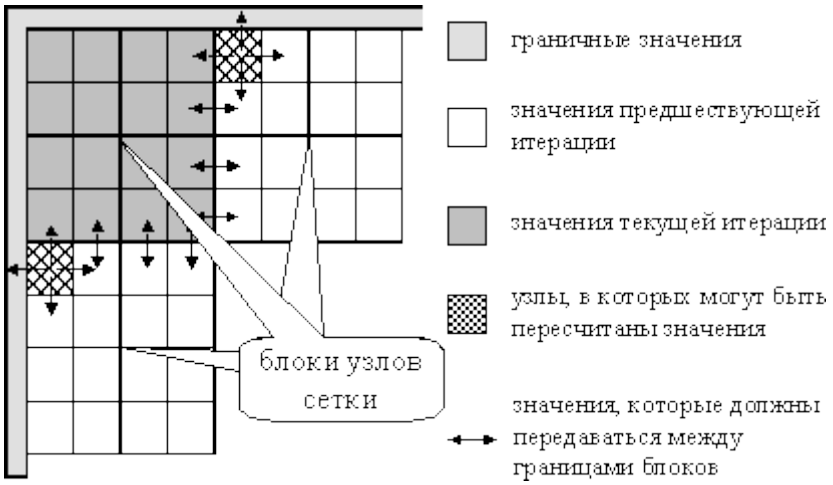

Рис. 6.9. Блочное представление сетки области расчетов

Порождаемый на основе такого подхода метод вычислений в самом общем виде может быть описан следующим образом (блоки образуют в области расчётов прямоугольную решётку размера NBxNB):

```
// Алгоритм 6.6
// 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 );
```

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. 6.3.

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 блоки 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 следует 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).

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, подбором значения для 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 учебного примера. На 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 6.7
// <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
```

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 [29, 31].

6.3. 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. раздел 1 пособия).

Многие проблемы параллельного программирования (состязание вычислений, тупики, 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). При проведении всех ниже рассмотренных экспериментов использовались 4 компьютера с процессорами Pentium IV, 1300 Mhz, 256 RAM, 100 Mbit Fast Ethernet.

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).

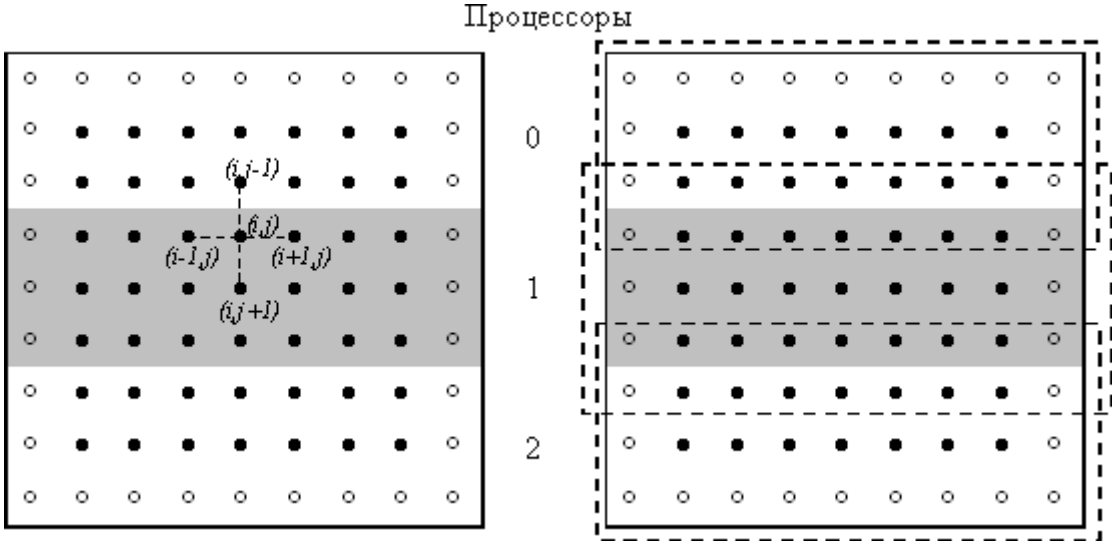

Рис. 6.10. Ленточное разделение области расчетов между процессорами (кружки представляют граничные узлы сетки)

В рассматриваемой учебной задаче по решению задачи Дирихле возможны два различных способа разделения данных – *одномерная* или *ленточная* схема (см. рис. 6.10) или *двухмерное* или *блочное* разбиение (см. рис. 6.9) вычислительной сетки. Дальнейшее изложение учебного материала будет проводиться на примере первого подхода; блочная схема будет рассмотрена позднее в более кратком виде.

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. 6.10 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.

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 6.8
// 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
```

- **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.

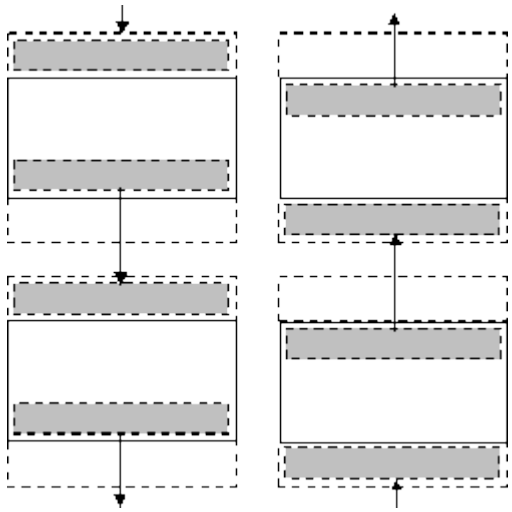

Рис. 6.11. Схема передачи граничных строк между соседними процессорами

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. 6.11). 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 [20] 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 – а*синхронная* 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. 6.11).

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 [21] 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.

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, полученный на отдельных полосах сетки, на один какой-либо процессор, вычисления на нем максимального значения и 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 п. 4.1 пособия *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 log2NP 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 [21] 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 6.8 – 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. 6.4.

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 6.4. Результаты экспериментов для систем с распределенной памятью, ленточная схема разделения данных (***p***=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 6.1)		Параллельный алгоритм 6.8			Параллельный алгоритм с волновой схемой расчета (см. п. 6.3.4)		
	k	***t***	***k***	***t***	***S***	***k***	***t***	***S***
100	210	0,06	210	0,54	0,11	210	1,27	0,05
200	273	0,35	273	0,86	0,41	273	1,37	0,26
300	305	0,92	305	0,92	1,00	305	1,83	0,50
400	318	1,69	318	1,27	1,33	318	2,53	0,67
500	343	2,88	343	1,72	1,68	343	3,26	0,88
600	336	4,04	336	2,16	1,87	336	3,66	1,10
700	344	5,68	344	2,52	2,25	344	4,64	1,22
800	343	7,37	343	3,32	2,22	343	5,65	1,30
900	358	9,94	358	4,12	2,41	358	7,53	1,32
1000	351	11,87	351	4,43	2,68	351	8,10	1,46
2000	367	50,19	367	15,13	3,32	367	27,00	1,86
3000	364	113,17	364	37,96	2,98	364	55,76	2,03

(***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. 6.12). 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. 6.4).

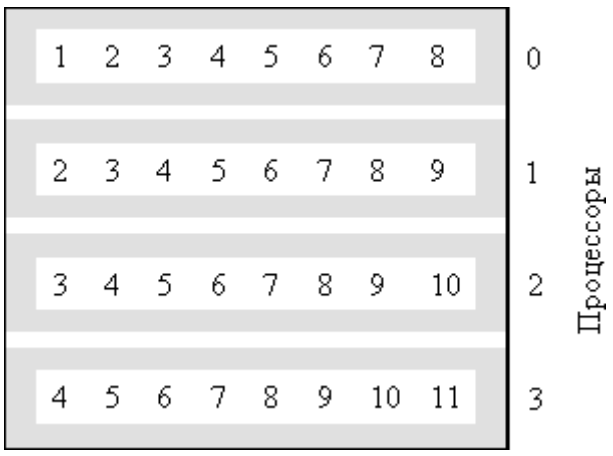

Рис. 6.12. Организация волны вычислений при ленточной схеме разделения данных

Интересной момент при организации подобный схемы параллельных вычислений может состоять в попытке совмещения операций пересылки граничных строк и действий по обработке блоков данных.

Блочная схема разделения данных

Ленточная схема разделения данных может быть естественным образом обобщена на блочный способ представления сетки области расчетов (см. рис. 6.9). 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 пересылается (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 количество 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. 6.5.

Таблица 6.5. Результаты экспериментов для систем с распределенной памятью, блочная схема разделения данных (***p***=4)

Размер сетки	Последовательный метод Гаусса-Зейделя (алгоритм 6.1)		Параллельный алгоритм с блочной схемой расчета (см. п. 6.3.5)			Параллельный алгоритм 6.9		
	k	***t***	***k***	***t***	***S***	***k***	***t***	***S***
100	210	0,06	210	0,71	0,08	210	0,60	0,10
200	273	0,35	273	0,74	0,47	273	1,06	0,33
300	305	0,92	305	1,04	0,88	305	2,01	0,46
400	318	1,69	318	1,44	1,18	318	2,63	0,64
500	343	2,88	343	1,91	1,51	343	3,60	0,80
600	336	4,04	336	2,39	1,69	336	4,63	0,87
700	344	5,68	344	2,96	1,92	344	5,81	0,98
800	343	7,37	343	3,58	2,06	343	7,65	0,96
900	358	9,94	358	4,50	2,21	358	9,57	1,04
1000	351	11,87	351	4,90	2,42	351	11,16	1,06
2000	367	50,19	367	16,07	3,12	367	39,49	1,27
3000	364	113,17	364	39,25	2,88	364	85,72	1,32

(***k*** – количество итераций, ***t*** – время в сек., ***S*** – ускорение)

При блочном представлении сетки может быть реализован также и волновой метод выполнения расчетов (см. рис. 6.13). Пусть процессоры образуют прямоугольную решетку размером $NBxNB$ ($NB=\sqrt{NP}$) и процессоры пронумерованы от 0 слева направо по строкам решетки.

Общая схема параллельных вычислений в этом случае имеет вид:

```
// Алгоритм 6.9
// 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
```

(в приведенном алгоритме функция **Barrier**() представляет операцию коллективной синхронизации, которая завершает свое выполнение только в тот момент, когда все процессоры осуществят вызов этой процедуры).

Следует обратить внимание, что при реализации алгоритма обеспечиться, 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. 6.13) и т.д.

Анализ эффективности организации волновых вычислений в системах с распределенной памятью (см. табл. 6.5) 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. 6.13), 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, получаемого при помощи исходного последовательного алгоритма.

Рис. 6.13. Организация волны вычислений при блочной схеме разделения данных

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 разделе 3 пособия.

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 свыше 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 – см. раздел 3 пособия. 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. 6.14):

- 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);

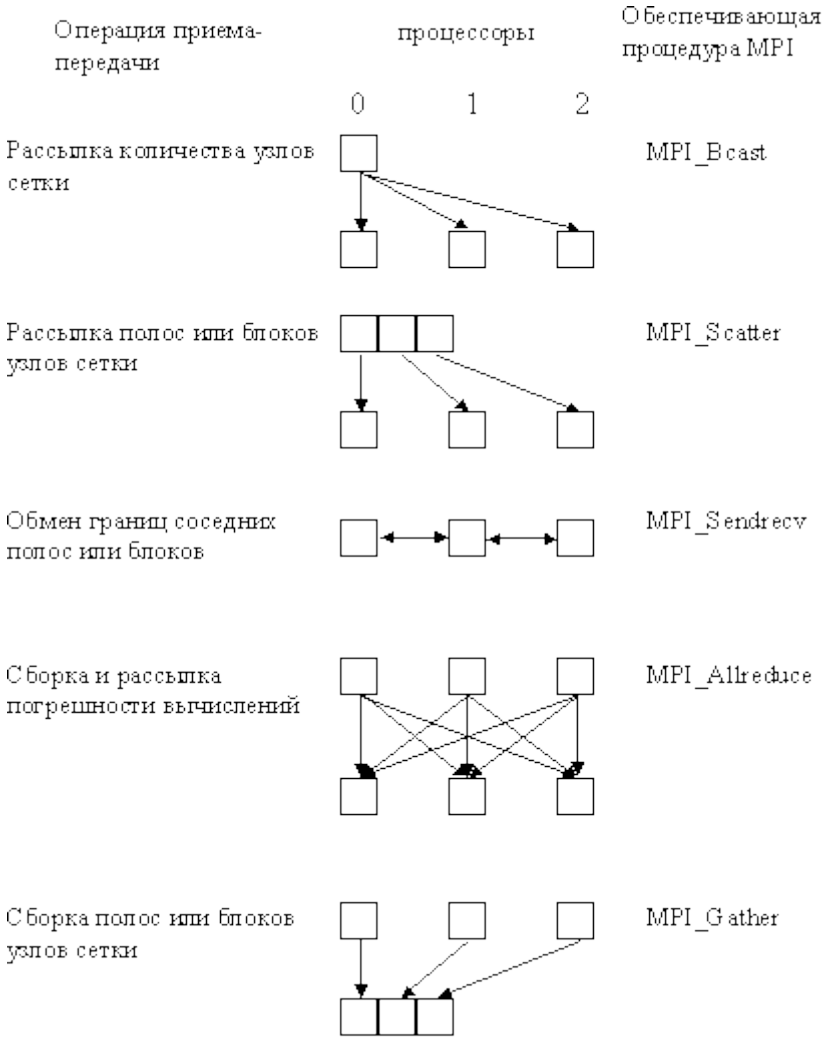

Рис. 6.14. Операции передачи данных при выполнении метода сеток в системе с распределенной памятью

- 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).

Литература

1. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. - Н.Новгород, ННГУ, 2001.
2. Богачев К.Ю. Основы параллельного программирования. - М.: БИНОМ. Лаборатория знаний, 2003.
3. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. - СПб.: БХВ-Петербург, 2002.
4. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем - СПб.: БХВ-Петербург, 2002.
5. Березин И.С., Жидков И.П. Методы вычислений. - М.: Наука, 1966.
6. Дейтел Г. Введение в операционные системы. Т.1.- М.: Мир, 1987.

7. Кнут Д. Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. - М.: Мир, 1981.

8. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. - М.: МЦНТО, 1999.

9. Корнеев В.В.. Параллельные вычислительные системы. - М.: Нолидж, 1999.

10. Корнеев В.В. Параллельное программирование в MPI. Москва-Ижевск: Институт компьютерных исследований, 2003.

11. П.Тихонов А.Н., Самарский А.А. Уравнения математической физики. -М.:Наука, 1977.

12. Хамахер К., Вранешич З., Заки С. Организация ЭВМ. - СПб: Питер, 2003.

13. Шоу А. Логическое проектирование операционных систем. - М.: Мир, 1981.

14. Andrews G.R. Foundations of Multithreading, Parallel and Distributed Programming. Addison-Wesley, 2000 (русский перевод Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. - М.: Издательский дом "Вильяме", 2003)

15. Barker, M. (Ed.) (2000). Cluster Computing Whitepaper http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/.

16. Braeunnl T. Parallel Programming. An Introduction.- Prentice Hall, 1996.

17. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. Parallel Programming in OpenMP. - Morgan Kaufinann Publishers, 2000

18. Dimitri P. Bertsekas, John N. Tsitsiklis. Parallel and Distributed Computation. Numerical Methods. - Prentice Hall, Englewood Cliffs, New Jersey, 1989.

19. Fox G.C. et al. Solving Problems on Concurrent Processors. - Prentice Hall, Englewood Cliffs, NJ, 1988.

20. 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.

21. Group W, Lusk E, Skjellum A. Using MPI. Portable Parallel Programming with the Message-Passing Interface. - MIT Press, 1994. (http://www.mcs.anl.gov/mpi/index.html)

22. Hockney R. W., Jesshope C.R. Parallel Computers 2. Architecture, Programming and Algorithms. - Adam Hilger, Bristol and Philadelphia, 1988. (русский перевод 1 издания: Р.Хокни, К.Джессхоуп. Параллельные ЭВМ. Архитектура, программирование и алгоритмы. - М.: Радио и связь, 1986)

23. Kumar V., Grama A., Gupta A., Karypis G. Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc., 1994

24. Miller R., Boxer L. A Unified Approach to Sequential and Parallel Algorithms. Prentice Hall, Upper Saddle River, NJ. 2000.

25. Pacheco, S. P. Parallel programming with MPI. Morgan Kaufmann Publishers, San Francisco. 1997.

26. Parallel and Distributed Computing Handbook. / Ed. A.Y. Zomaya. -McGraw-Hill, 1996.

27. Pfister, G. P. In Search of Clusters. Prentice Hall PTR, Upper Saddle River, NJ 1995. (2nd edn., 1998).

28. Quinn M. J. Designing Efficient Algorithms for Parallel Computers. - McGraw-Hill, 1987.

29. Rajkumar Buyya. High Performance Cluster Computing. Volume l: Architectures and Systems. Volume 2: Programming and Applications. Prentice Hall PTR, Prentice-Hall Inc., 1999.

30. Roosta, S.H. Parallel Processing and Parallel Algorithms: Theory and Computation. Springer-Verlag, NY. 2000.

31. Xu, Z., Hwang, K. Scalable Parallel Computing Technology, Architecture, Programming. McGraw-Hill, Boston. 1998.

32. Wilkinson B., Allen M. Parallel programming. - Prentice Hall, 1999.

Информационные ресурсы сети Интернет

33. Информационно-аналитические материалы по параллельным вычислениям (http://www.parallel.ru)

34. Информационные материалы Центра компьютерного моделирования Нижегородского университета (http://www.software.unn.ac.ru/ccam)

35. Информационные материалы рабочей группы IEEE по кластерным вычислениям (http://www.ieeetfcc.org)

36. Introduction to Parallel Computing (Teaching Course) (http://www.ece.nwu.edu/~choudhar/C58/)

37. Foster I. Designing and Building Parallel Programs. - Addison Wesley, 1994. (http://www.mcs.anl.gov/dbpp)