МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

В.А. ФУРСОВ

ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА КЛАСТЕРЕ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)» в качестве учебного пособия для студентов, обучающихся по программам высшего образования по направлениям подготовки бакалавров 01.03.02 Прикладная математика и информатика, 09.03.01 Информатика и вычислительная техника

САМАРА
Издательство СГАУ
2015

УДК 004(075)+004.932(075)
ББК 32.973я7
Ф 954

Рецензенты: д-р техн. наук, доц. С.Б. Попов,
д-р техн. наук, доц. С.А. Востокин

Фурсов В.А.
Ф 954 **Введение в параллельные вычисления на кластере**: учеб. пособие / *В.А. Фурсов*. – Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2015. – 80 с.

ISBN 978-5-7883-1041-1

В учебном пособии излагаются основы построения параллельных алгоритмов и программ. Цель учебного пособия – помочь студентам младших курсов в максимально короткие сроки освоить технологию решения задач на многопроцессорной высокопроизводительной вычислительной системе. Кратко излагаются основные сведения, необходимые для построения параллельных алгоритмов, приводятся минимальные сведения о библиотеке MPI, пример простой параллельной программы и краткая инструкция для запуска параллельных приложений на кластере СГАУ «Сергей Королёв».

Ориентировано на подготовку студентов машиностроительных факультетов, в учебных планах которых на курс «Информатика» отводится ограниченное число часов. Может быть полезно также для других категорий учащихся, в том числе школьников, приступающих к освоению технологий параллельных вычислений.

Предназначено для студентов, обучающихся по программам высшего образования по направлениям подготовки бакалавров 01.03.02 Прикладная математика и информатика, 09.03.01 Информатика и вычислительная техника.

УДК 004(075)+004.932(075)
ББК 32.973я7

ISBN 978-5-7883-1041-1

ОГЛАВЛЕНИЕ

Введение

В последние годы возрастают требования к подготовке студентов различных специальностей и направлений в области информатики. В частности, в учебные программы курса «Информатика» прочно вошли разделы, посвященные вычислениям на высокопроизводительных вычислительных системах. Связано это с качественно новым уровнем требований к инженерам технических специальностей. Современный конструктор и технолог должен иметь уверенные навыки моделирования элементов и конструкций изделий на различных этапах проектирования. Для этого необходимо проведение масштабных численных экспериментов на высокопроизводительных многопроцессорных вычислительных системах.

Подготовка задачи к решению на параллельном компьютере заключается в составлении плана вычислений и написании кода. Первое обычно требует глубокого понимания существа задачи, а второе – знания основ программирования и понимания особенностей построения вычислительных систем. В частности, требуется знание топологии связей, типа процессора, относительных объемов памяти кэшей различных уровней и др.

Подготовка узких специалистов высокой квалификации в области высокопроизводительных вычислительных технологий в настоящее время ведется в ряде учебных заведений на специализированных факультетах. Например, в СГАУ таких специалистов готовит факультет информатики. Тем не менее, студентам инженерных факультетов, специализирующимся в области конструирования двигателей и летательных аппаратов, также необходима достаточно серьезная подготовка в области высокопроизводительных вычислений. Инженер-конструктор должен уметь поставить задачу разработки параллельной вычислительной программы по крайней мере на уровне модели параллельных вычислений. Для этого он должен понимать общие принципы формирования этой модели с учетом ее дальнейшей программной реализации на многопроцессорной системе.

К сожалению, подготовка специалистов, одинаково глубоко владеющих как предметом исследований в «своей» области знаний, так и знаниями в области высокопроизводительных вычислительных систем, представляется сложной проблемой. В последние годы издается немало монографий и учебников, например [2, 3], которые могут использоваться при подготовке студентов. Большинство этих изданий посвящено детальному рассмотрению достаточно сложных вопросов построения параллельных алгоритмов и программ, изучение которых в рамках общего курса информатики на инженерных факультетах не представляется возможным. Массовая подготовка инженеров, в особенности при переходе на двухступенчатую подготовку бакалавр-магистр (с крайне малым числом часов аудиторных занятий), потребовала издания компактного и вместе с тем достаточно доступного для самостоятельной подготовки пособия.

Цель пособия – помочь студентам, имеющим лишь начальную подготовку в области информатики, в максимально короткие сроки освоить технологию решения задач на многопроцессорной высокопроизводительной вычислительной системе. В соответствии с поставленной целью стиль конспективный, возможно, в ущерб строгости изложения материала. В частности, в максимально доступной форме даются лишь минимально необходимые сведения, касающиеся вопросов анализа параллельной структуры алгоритмов, построения моделей и схем параллельных вычислений, а также методов анализа эффективности решения вычислительных задач.

Изучение этих вопросов сопровождается простым примером параллельного алгоритма и соответствующей параллельной программы. Этот пример поможет начинающему пользователю суперкомпьютера перейти к подготовке более сложных параллельных приложений. Обычно это не вызывает затруднений, т.к. принципы построения многих параллельных программ сходны. Подчеркнем, что большая часть учебного пособия посвящена вопросам построения эффективных моделей параллельных вычислений, которые затем используются на этапе написания кода параллельной программы. К сожалению, этим важным аспектам часто не уделяют должного внимания даже опытные программисты.

Учебное пособие в значительной степени представляет собой методически переработанное учебное пособие [1] с ориентацией на начальную подготовку студентов машиностроительных специальностей, в учебных планах которых на подготовку по курсу «Информатика» отводится ограниченное число часов. Поэтому все доказательства и сложные вопросы, касающиеся детализации и обоснования некоторых утверждений, вынесены в приложения.

Студенты первого курса при первом прочтении могут опустить изучение этих приложений. Вместе с тем их наличие открывает возможность более углубленного изучения этих вопросов в дальнейшем. Кроме того, это делает учебное пособие полезным также для слушателей ФПКП и других категорий учащихся, приступающих к освоению технологий параллельных вычислений, не имеющих достаточно глубокой подготовки в области информационных технологий, но обладающих достаточной общеинженерной подготовкой.

Автор выражает признательность Гошину Егору Вячеславовичу, который подготовил и отладил параллельную программу и отредактировал заключительный раздел.

1. Краткая история развития суперкомпьютеров

Первая супер-ЭВМ CDC-6600 была разработана Сеймуром Креем (Cray, Seimour; 1925–1996) и построена в фирме Control Data Corporation (1963 г.). Разрядность – 64 бита, быстродействие – 3 млн. оп./с. Однако одновременно на качественно новый уровень вышел и рынок мини-ЭВМ. Наиболее известные решения в этом направлении: мини-ЭВМ PDP: -5 (1963), -8 (1965) -11 (1970): первые модели ПК компаний MITS и Apple (Altair-1975, Apple-II, 1977); IBM-совместимые ПК (IBM PC XT – 1983, PC AT – 1984); Apple: Macintosh (1984).

Появление на рынке сравнительно дешевых и достаточно высокопроизводительных персональных компьютеров, с одной стороны, и развитие высокопроизводительных сетевых технологий, с другой стороны, обусловило появление высокопроизводительных вычислительных систем – кластеров.

Основой достижений в обоих указанных направлениях развития высокопроизводительных вычислительных систем являются успехи в развитии элементной базы. Ниже приводится график, иллюстрирующий так называемый закон Мура (рис. 1.1), согласно которому количество элементов на одном кристалле удваивается каждые полтора года. Несмотря на то, что уже ощущается предел, обусловленный физическими закономерностями, этот закон пока выполняется.

Бытует мнение, что в России и Советском Союзе не было хорошей вычислительной техники. В действительности это не так. В развитии отечественной вычислительной техники был период расцвета (1950–1960-е гг.), который, к сожалению, был прерван не без вмешательства руководства страны на самом высоком уровне.

В 1950 году академик М.А. Лаврентьев написал записку И.В. Сталину, в которой обосновал важность создания высокопроизводительных вычислительных машин. Реакцией на это письмо стали подготовка постановления правительства в удивительно короткие сроки и начало разработки ЭВМ одновременно в Академии наук СССР и Министерстве машиностроения и приборостроения. В результате уже в 1951 году под руководством С.А. Лебедева появилась

первая отечественная ЭВМ МЭСМ (1951 г., Киев). Она имела следующие характеристики: 6000 электронных ламп, быстродействие 50 оп./с, ОЗУ 94 16-разрядных слов, потребляемая мощность – 15 кВт, занимаемая площадь – 60 кв.м. В 1952 г. была создана уже полномасштабная отечественная ЭВМ БЭСМ со следующими характеристиками: 5000 ламп, быстродействие 8000 оп./с, ОЗУ-1К 39-разрядных слов, ПЗУ-1К слов, потребляемая мощность – 30 кВт, занимаемая площадь – 100 кв. м.

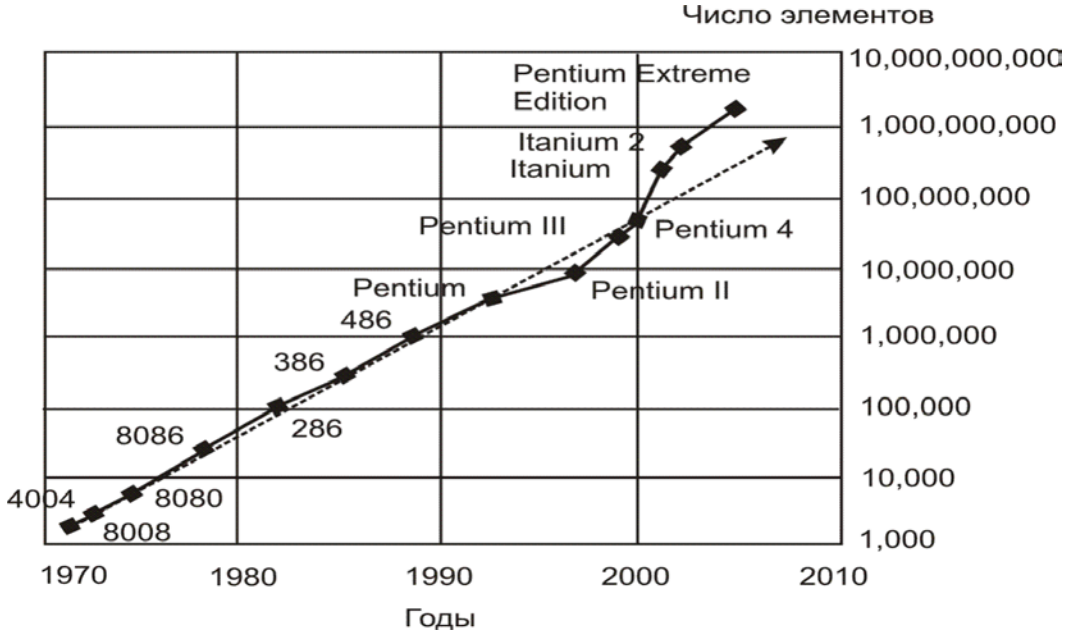


Рис. 1.1. Закон Мура

Для сопоставления темпов развития вычислительной техники в России и за рубежом приводится табл. 1.1, в которую включены некоторые наиболее успешные решения. Из таблицы видно, что отставание в производительности имело место, но оно не было катастрофическим. Объективно перелом произошел лишь в результате микропроцессорной революции. Субъективным фактором стало решение советского руководства (в конце 1960-х годов) о прекращении производства оригинальных отечественных ЭВМ и развертывании работ по созданию Единой системы ЭВМ (ЕС ЭВМ) социалистических стран на базе архитектуры IBM System/360, а также Системы малых машин (СМ ЭВМ) на базе архитектуры Hewlett Packard и PDP-11. Всего за период с 1970 по 1990 год было произведено более 15000 вычислительных машин разной производительности ряда ЕС ЭВМ.

Таблица 1.1. Сопоставление отечественных и зарубежных ЭВМ

Год	ЭВМ в СССР	Зарубежный образец
1951	МЭСМ. 50 оп./с, ОЗУ – 94 16-разрядных слова	UNIVAC-1. 2000 оп./с, ОЗУ – 1000 слов по 12 десятичных разрядов
1952	БЭСМ. 5000 ламп, 8000 оп./с, ОЗУ – 1К 39-разрядных слов, ПЗУ – 1К слов	IBM-701. 10000 оп./с, ОЗУ – 2К 36-разрядных слов (1952)
1953	«Стрела». 6200 ламп, 60000 п/п диодов. 2000 оп./с	IBM-702
1958	М-20. 2600 ламп, ОЗУ – 4К 45-разрядных слов, 20 000 оп./с	IBM-7030 Stretch (1959 г.), 500 тыс. оп./с, ОЗУ – до 256К 64-битовых слов
1964	Урал - 11, - 14, -16 45 до 100 тыс. оп./с	IBM System/360 (1964). Возможно создание серии вычислительных машин различной мощности. Появление чипов с числом 10 (1964) и 100 (1970) элементов. Создание чипов с числом транзисторов до 65 тыс. (1975)
1968	БЭСМ-6. 60 тыс. транзисторов, 180 тыс. диодов,1 млн. оп./с, ОЗУ – от 32К до128К 48-разрядных слов	
1968	«Минск-32». 30-35 тыс. оп./с, ОЗУ – 64К 38-разрядных слов	

Несмотря на огромные затраты, качественного роста вычислительной техники в Советском Союзе не произошло. Технологическая, а как следствие, и элементная база оставались слабыми. Были разрушены сложившиеся научные школы, способные создавать конкурентоспособные вычислительные машины. Наиболее отрицательным последствием копирования стала возросшая зависимость от заимствованного программного обеспечения. Эта зависимость остро ощущается до сих пор, что сдерживает создание и реализацию сложных систем во многих областях.

В заключение краткого введения в историю развития отечественной высокопроизводительной вычислительной техники подчеркнем, что период успехов в развитии вычислительной техники (50-е годы) удивительным образом совпадает с впечатляющими успехами СССР в других областях: космических исследованиях, развитии атомной энергетики и др.

В последние годы в России вновь ощущается потребность в ускоренном развитии ключевых отраслей экономики и связанного с этим направления, кратко обозначаемого как суперкомпьютерные вычисления. Это направление включает как создание суперкомпьютерных вычислительных систем, так и разработку системного и прикладного программного обеспечения. В связи с этим в последние годы возрос интерес к освоению суперкомпьютерных вычислений и, как следствие, потребность в учебных пособиях, дающих возможность сделать это в короткие сроки.

2. Классификация алгоритмов по типу параллелизма

В настоящее время ощущается, что дальнейшее повышение производительности компьютеров только за счет улучшения характеристик элементов электроники, в рамках традиционных технологий, ограничивается физическими пределами свойств материалов. Поэтому дальнейшее повышение производительности возможно лишь за счет распараллеливания процессов обработки информации.

Идея сокращения времени выполнения большого объема работ путем разбиения на отдельные работы, которые могут выполняться *независимо* и *одновременно,* не нова и используется во многих отраслях (строительстве, машиностроении и др.). В вычислительных машинах это достигается путем увеличения числа одновременно работающих процессоров. При этом возникает естественное желание обычную последовательную программу, которая ранее уже запускалась на однопроцессорном компьютере, перенести на многопроцессорную вычислительную систему. Как ни странно, это может не дать выигрыша в производительности, более того, часто в результате такого переноса программа работает медленнее.

При переходе к использованию многопроцессорных систем необходимо провести структурный анализ алгоритма и выявить потенциальные возможности его распараллеливания. Способность алгоритма к распараллеливанию потенциально связана с одним из двух (или одновременно с обоими) внутренних свойств, которые характеризуются как *параллелизм задач* (message passing) и *параллелизм данных* (data parallel).

Если алгоритм основан на параллелизме задач, вычислительная задача разбивается на несколько относительно самостоятельных подзадач, каждая из которых загружается в «свой» процессор. Каждая подзадача реализуется независимо, но использует общие данные и/или обменивается результатами своей работы с другими подзадачами. Для реализации такого алгоритма на многопроцессорной системе необходимо выявлять независимые подзадачи, которые могут выполняться параллельно. Часто это оказывается далеко не очевидной и весьма трудной проблемой.

При наличии в алгоритме свойства параллелизма данных одна операция может выполняться сразу над всеми элементами исходного массива данных. В этом случае различные фрагменты массива могут обрабатываться независимо на разных процессорах. Для алгоритмов этого типа распределение данных между процессорами обычно осуществляется до выполнения задачи на ЭВМ. Построение алгоритма, обладающего свойством параллелизма данных, и подбор подходящей архитектуры компьютера для него могут выполняться с использованием достаточно простых методик, не требующих применения сложного математического аппарата.

Для того чтобы в полной мере использовать структурные свойства алгоритма, необходимо выявить, к какому типу он относится. Ниже приводится общая классификация алгоритмов с точки зрения типа параллелизма [10].

1. Алгоритмы с распределением данных (Data Partitioning). Это алгоритмы решения задач, в которых пространство данных может быть разделено на непересекающиеся области, с каждой из которых связаны независимые процессы, оперирующие каждый со своими данными.

2. Алгоритмы, использующие параллелизм данных (Data Parallelism). Этот тип параллелизма характерен для численных алгоритмов обработки данных, которые также могут быть разделены на непересекающиеся области, обработка которых может осуществляться независимо, но требуется обмен данными между параллельными процессами.

3. Алгоритмы с синхронизацией итераций (Synchronous Iteration). Синхронизация в конце каждой итерации заключается в том, что разрешение на начало следующей итерации дается после того, как все процессоры завершили предыдущую итерацию. Обычно к таким алгоритмам приводятся задачи, в которых исходные данные могут быть разделены на области с некоторым перекрытием, при этом возникает необходимость обмена данными на границах.

4. Релаксационные алгоритмы (Relaxed Algorithm). В отличие от предыдущего алгоритм может быть представлен в виде независимых процессов без синхронизации связи между ними, но процессоры должны иметь доступ к общим данным.

5. Самовоспроизводящиеся задачи (Replicated Workers). Для задач этого класса создается и поддерживается центральный пул (хранилище) похожих вычислительных задач. Параллельно реализуемые процессы осуществляют выбор задач из пула, выполнение вычислений и добавление новых задач к пулу. Вычисления заканчиваются, когда пул пуст. Эта технология характерна для исследований графа или дерева.

6. Конвейерные вычисления (Pipelined Computation). Этот тип вычислений характерен для процессов, которые могут быть представлены в виде некоторой регулярной структуры, например, в виде кольца или двумерной сети. Каждый процесс, находящийся в узле этой структуры, реализует определенную фазу вычислений.

Нетрудно заметить, что некоторые алгоритмы из приведенного списка обладают явно выраженными свойствами параллелизма задач или параллелизма по данным. Вместе с тем ряд алгоритмов в той или иной мере обладают *обоими* указанными свойствами. В ходе структурного анализа должны быть выявлены типы параллелизма и выбрана схема формирования параллельного алгоритма.

Общий подход к построению параллельных алгоритмов заключается в их представлении в виде одного или нескольких взаимодействующих процессов. Процессы – это логически завершенные элементарные работы, на которые можно и целесообразно разбить выполняемый алгоритм. Структура общего ал-

горитма должна устанавливать правила взаимодействия процессов. При формировании процессов основная проблема – выделение элементарных работ, выполнение которых в вычислительной системе подлежит распараллеливанию.

С точки зрения простоты организации параллельных вычислений, конечно, удобнее делить задачу на крупные подзадачи, каждая из которых решается на «своем» процессоре. Однако при этом будет задействовано небольшое число процессоров и добиться высокого ускорения не удастся. Выбор степени «измельчения» задачи и обоснование числа параллельных процессоров является трудно формализуемым, в значительной степени творческим этапом, и его успешность часто зависит от наличия у пользователя опыта решения подобных задач.

Часто существенное повышение эффективности параллельной реализации алгоритма достигается путем коренного пересмотра математической формулировки задачи. Следует иметь в виду, что наибольший эффект достигается, если при формулировке задачи сразу принимается во внимание конкретная архитектура многопроцессорной системы, на которой предполагается ее решение. Такая формулировка не всегда очевидна, однако усилия в этом направлении всегда оправданы, т.к. параллельные алгоритмы, построенные с учетом конкретной архитектуры параллельной ЭВМ, как правило, дают наибольшее ускорение и эффективность.

Поэтому пользователю, приступающему к решению сложной в вычислительном отношении задачи, прежде всего целесообразно познакомиться с архитектурой вычислительной системы, на которой предполагается реализация параллельной программы. Ниже приводится краткий обзор типовых архитектур вычислительных систем.

3. Архитектура вычислительных систем

Для того чтобы подобрать вычислительную систему, на которой может быть эффективно реализован конкретный параллельный алгоритм, или построить вычислительную схему, наиболее подходящую для имеющейся в распоря-

жении пользователя вычислительной системы, необходимо ясно представлять потенциальные возможности различных архитектур. Рассмотрим некоторые наиболее известные классификации и особенности построения компьютерных систем.

При разработке параллельного алгоритма наиболее важным фактором является тип оперативной памяти. В зависимости от организации подсистем оперативной памяти параллельные компьютеры можно разделить на следующие два класса.

1. Системы с разделяемой памятью (*мультипроцессоры*), у которых имеется одна виртуальная память, а все процессоры имеют одинаковый доступ к данным и командам, хранящимся в этой памяти (uniform memory access, или UMA). По этому принципу строятся векторные параллельные процессоры (parallel vector processo, или PVP) и симметричные мультипроцессоры (symmetric multiprocessor, или SMP), которые состоят из совокупности процессоров, имеющих разделяемую общую память с единым адресным пространством и функционирующих под управлением одной операционной системы (рис. 3.1).

Рис. 3.1. Архитектура многопроцессорных систем с общей (разделяемой) памятью

Конечно, самый простой способ коммутации процессоров – использование общей шины. В данном случае пользователю не нужно заботиться о распределении данных, достаточно лишь один раз задать соответствующую структуру данных в оперативной памяти и построить процедуру выбора необходимых данных из этой памяти в процессе решения задачи. Однако в таких системах даже небольшое увеличение числа процессоров, подключаемых к общей шине,

делает ее узким местом. Поэтому компьютеры этого класса обычно имеют небольшое число процессоров, т.к. наращивание числа процессоров ведет к быстрому росту потерь на межпроцессорный обмен данными.

2. Системы с распределенной памятью (*мультикомпьютеры*), у которых каждый процессор имеет свою локальную оперативную память, а у других процессоров доступ к этой памяти отсутствует. Этот класс систем часто называют также *массово-параллельными* (или *массивно-параллельными*) компьютерами.

Различия компьютеров данного класса в основном сводятся к различиям в организации коммуникационной среды. Известны архитектуры, в которых процессоры расположены в узлах прямоугольной решетки. Иногда взаимодействие идет через иерархическую систему коммутаторов, обеспечивающих возможность связи каждого узла с каждым. Используется также топология трехмерного тора, т.е. каждый узел имеет шесть непосредственных соседей вне зависимости от того, где он расположен. В главе 5 мы приведем некоторые наиболее известные типовые схемы коммуникационных сетей.

На рис. 3.2 показана общая схема связей основных элементов системы в архитектуре многопроцессорных систем с распределенной памятью. Преимущество этой архитектуры – возможность практически неограниченного наращивания числа процессоров. Для систем этого класса характерно также низкое значение отношения цена/производительность. При работе на компьютере с распределенной памятью необходимо создавать копии исходных данных на каждом процессоре. Поэтому наиболее подходящими для реализации на системах данного класса являются алгоритмы *с распределением данных*, в которых пространство данных декомпозируется без пересечений, а каждая область может обрабатываться отдельно без обмена данными с другими.

Однако часто это вызывает серьезные трудности, связанные с недостаточным объемом памяти в узлах. Поэтому при реализации параллельного алгоритма на компьютерах с распределенной памятью важно построить рациональную с точки зрения потерь на обмен и допустимых объемов памяти в узлах схему размещения данных.

Рис. 3.2. Архитектура многопроцессорных систем с распределенной памятью

Компромисс между системами с *разделяемой* и *распределенной* памятью достигается с использованием программно-аппаратных решений, реализующих неоднородный доступ к памяти (non-uniform memory access, или NUMA). При такой реализации общая память является физически распределенной, однако все процессоры имеют доступ к памяти любого процессора (*распределенная общая память*). Это позволяет увеличить число процессоров, но сохранить возможность работы в рамках единого адресного пространства при приемлемых потерях на межпроцессорный обмен. Основная проблема, которую решают в рамках этого подхода, – обеспечение когерентности кэш-памяти отдельных процессоров. Эта трудность преодолевается применением специальных программно-аппаратных средств. На рис. 3.3 приведен пример схемы связей элементов в мультипроцессорных системах с неоднородным доступом к разделяемой памяти.

Использование распределенной общей памяти (distributed shared memory, или DSM) упрощает проблемы создания параллельных программ. При этом в системе может функционировать огромное число параллельных процессоров, однако время доступа к локальной и удаленной памяти может существенно различаться. В данном случае необходимо планировать распределение процессоров и схему обмена данными так, чтобы минимизировать число обращений к удаленной памяти. Это может быть учтено на этапе составления расписания

параллельного алгоритма. Соответствующая методика будет описана в следующих разделах. Заметим также, что системы с разделяемой памятью обычно имеют высокую стоимость.

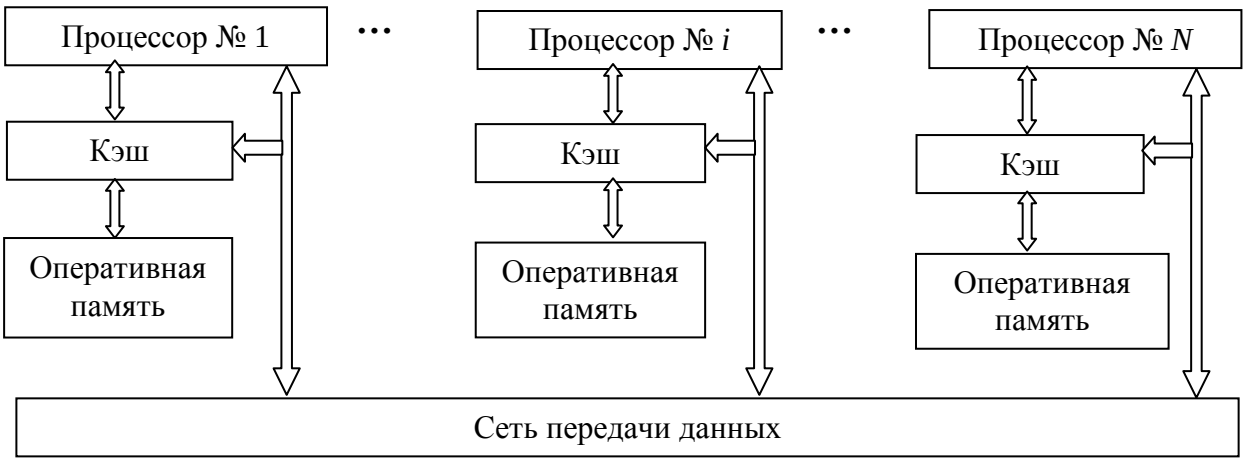


Рис. 3.3. Архитектура мультипроцессорной системы
с неоднородным доступом к разделяемой памяти

Известна также другая классификация компьютеров, основанная на понятии *потока*, под которым понимается последовательность команд или данных, обрабатываемых процессором:

- SISD (Single Instruction stream/Single Data stream) – один поток команд и один поток данных;
- SIMD (Single Instruction stream/Multiple Data stream) – один поток команд и множество потоков данных;
- MISD (Multiple Instruction stream/Single Data stream) – множество потоков команд и один поток данных;
- MIMD (Multiple Instruction stream/Multiple Data stream) – множество потоков команд и множество потоков данных.

Класс SISD являлся характерным для архитектуры ранних компьютеров. Современные компьютеры чаще строятся на основе MIMD-архитектуры или с использованием комбинации нескольких архитектур в одной системе.

SIMD – *векторно-конвейерные* компьютеры, в которых используются векторные команды, обеспечивающие выполнение операций с массивами незави-

симых данных за один такт. Типичным представителем данного направления является линия «классических» векторно-конвейерных компьютеров CRAY. Собственно появление термина *суперкомпьютер* связано именно с созданием высокопроизводительного компьютера Cray-1 (1976) с *векторной* архитектурой.

Эффективность использования векторной архитектуры зависит от наличия в алгоритме большого числа одинаковых и независимых операций. Если же вычисление некоторого элемента массива не может начаться, пока не будет вычислен предыдущий элемент, такой алгоритм не векторизуется. Другими словами, *зависимость между операциями* всегда препятствует векторизации. В векторной программе операции выполняются над всеми элементами регистра. В параллельной программе каждый из процессоров выполняет команды, оперируя со своими собственными регистрами. В обоих случаях действия выполняются одновременно, однако каждый из процессоров параллельной ЭВМ может реализовывать алгоритм, отличающийся от алгоритмов других процессоров. Другими словами, алгоритм, который векторизуется, всегда можно и распараллелить. Обратное утверждение не всегда верно.

Кластеры образуются из вычислительных узлов, объединенных системой связи или посредством разделяемой внешней памяти. Кластерные проекты связаны с появлением на рынке недорогих микропроцессоров и коммуникационных решений. В результате появилась реальная возможность создавать установки «суперкомпьютерного» класса из составных частей массового производства. При этом могут использоваться как специализированные, так и универсальные сетевые технологии, которые обычно и определяют эффективность кластерных решений.

Для характеристики сетей в кластерных системах используют два параметра: латентность и пропускную способность. *Латентность* – это время начальной задержки при посылке сообщений. *Пропускная способность сети* определяется скоростью передачи информации по каналам связи. Если алгоритм содержит много коротких сообщений, то наиболее критической характеристикой

является латентность. Если передача сообщений организована большими пакетами данных, более важной является пропускная способность.

Для повышения эффективности реализации параллельной программы на кластере необходимо обеспечить *равномерную загрузку всех процессоров*. Если вычислительная система неоднородна, а объемы вычислительных работ распределены между процессорами равномерно, то часть процессоров может простаивать.

В принципе, любые вычислительные устройства можно считать параллельной вычислительной системой, если они работают одновременно и их можно использовать для решения одной задачи. Под это определение попадают и компьютеры в сети Интернет. Интернет можно рассматривать как самый мощный кластер – метакомпьютер. Процесс организации вычислений в такой вычислительной системе – *метакомпьютинг*. Интерес к метакомпьютингу в последние годы возрастает в связи с необходимостью обработки огромных объемов информации.

4. Построение параллельных алгоритмов, инженерный подход

Разработку параллельного алгоритма обычно осуществляет (по крайней мере начинает) специалист, работающий в некоторой предметной области. При этом чаще всего он вынужден ориентироваться на доступную ему вычислительную систему с некоторой сложившейся архитектурой. Поэтому естественно стремление находить некоторое приемлемое решение, руководствуясь не вполне строгими, но проверенными на практике приемами и правилами. В частности, если решаемая задача обладает свойством *параллелизма по данным*, то эти правила обычно очевидны и позволяют строить весьма эффективные параллельные алгоритмы.

Совокупность выработанных на основе опыта и здравого смысла методов и приемов распараллеливания задач, обладающих свойством декомпозируемости по данным, будем называть *инженерным* подходом. В данном случае суще-

ство дела сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Основной целью построения параллельной программы в этом случае является обеспечение заданной степени загрузки всех процессоров. Для этого, с одной стороны, необходимо обеспечить равномерную загрузку процессоров, возможно, с учетом их различной производительности. С другой стороны, необходимо обеспечить допустимое соотношение временных затрат на пересылку данных (накладные расходы) и проведение вычислений на фрагментах исходных данных [2, 3].

Для реализации указанных целей при построении параллельного алгоритма в рамках инженерного подхода обычно решаются следующие задачи [3]:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо.
2. Определение для этого набора подзадач информационных взаимодействий.
3. Масштабирование подзадач, определение количества процессоров.
4. Закрепление подзадач за процессорами, составление расписания.

Характер и последовательность указанных действий могут различаться в зависимости от архитектуры системы, числа доступных процессоров и др. Если число подзадач (областей данных) меньше числа доступных процессоров, выполняют *масштабирование* параллельного алгоритма, которое сводится к декомпозиции первоначально заданных областей данных. Для сокращения количества подзадач укрупняют области исходных данных, притом в первую очередь объединяют области, для которых соответствующие подзадачи обладают высокой степенью информационной взаимозависимости.

Распределение подзадач между процессорами очевидно, если количество областей данных совпадает с числом доступных процессоров, а топология сети передачи данных – полный граф, т.е. все процессоры связаны между собой. Если это не так, то подзадачи, имеющие информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных. Требование минимизации информационных обменов между процессорами может вступить в противоречие с условием равномерной загрузки. Решение вопросов балансировки вычислительной нагрузки значи-

тельно усложняется, если схема вычислений изменяется в ходе решения задачи. При этом целесообразна динамическая балансировка в ходе выполнения программы.

Способ разделения вычислений на независимые части зависит от того, в какой степени решаемая задача обладает свойством декомпозируемости по данным, другими словами, какое место занимает алгоритм в классификации, приведенной в главе 2.

Простейший и наиболее благоприятный с точки зрения организации параллельных вычислений случай, когда вся область исходных данных задачи может быть разделена на непересекающиеся области любых размеров, а вычисления в каждой области могут вестись независимо (*алгоритмы с распределением данных*). В этом случае задача построения параллельного алгоритма проста: необходимо всю область разбить на подобласти, число которых равно числу доступных процессоров, а размеры подобластей подобрать так, чтобы обеспечить их равномерную загруженность с учетом производительности каждого.

Для большинства практических задач при декомпозиции по данным вычисления в каждой области не могут быть полностью независимыми. В частности, после каждой итерации (проведения вычислений во всех точках области) возникает потребность обмена результатами вычислений на границах соседних областей. Это, например, характерно для большинства сеточных методов, в которых для вычисления значения функции в некотором узле используются ее значения в нескольких соседних узлах. В этом случае требование равномерной вычислительной загрузки процессоров становится еще более жестким. При неравномерной загрузке процессоров часть из них может простаивать в ожидании завершения обмена данными.

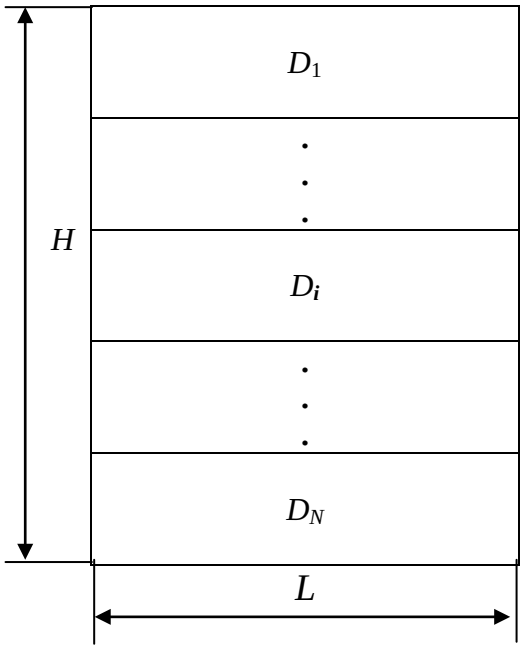


Рис. 4.1. Ленточная схема декомпозиции

Сбалансированность загрузки процессоров и минимизация информационных обменов зависит не только от размеров подобластей, но также и от их формы. С точки зрения организации вычислений обычно более удобной является декомпозиция на области с границами в виде прямых линий и плоскостей. В случае двумерной задачи наиболее часто используется один из двух типов декомпозиции – разделение на отдельные строки или последовательные группы строк (ленточная схема разделения данных), разделение на прямоугольные наборы элементов (блочная схема разделения данных). На рис. 4.1 показана ленточная схема декомпозиции, а на рис. 4.2 – блочная. Возникает естественный вопрос: какая из этих схем декомпозиции «лучше»?

Выбор одной из указанных схем декомпозиции диктуется требованием минимизации пересылок данных между процессорами. Рассмотрим эту задачу для двумерного случая. Будем полагать, что области заданы в виде прямоугольников или квадратов, ширина полос данных в окрестности границ, которыми должны обмениваться процессоры, не зависит от направления границ фрагментов, а объем передаваемых данных определяется длиной сопряженных границ фрагментов. Приведем простой пример декомпозиции двумерной области, имеющей размеры $H \times L$, $H \geq L$.

При декомпозиции области данных на четыре подобласти (процессора) общий объем передаваемых данных при ленточном разделении данных (вдоль стороны L) пропорционален $3L$, а при блочном (на равные прямоугольники) – $H + L$. Объем максимального межпроцессорного обмена данными между парами процессоров, обрабатывающих соседние области, составит соответственно L и $H/2$. Нетрудно заметить, что одинаковый общий и максимальный

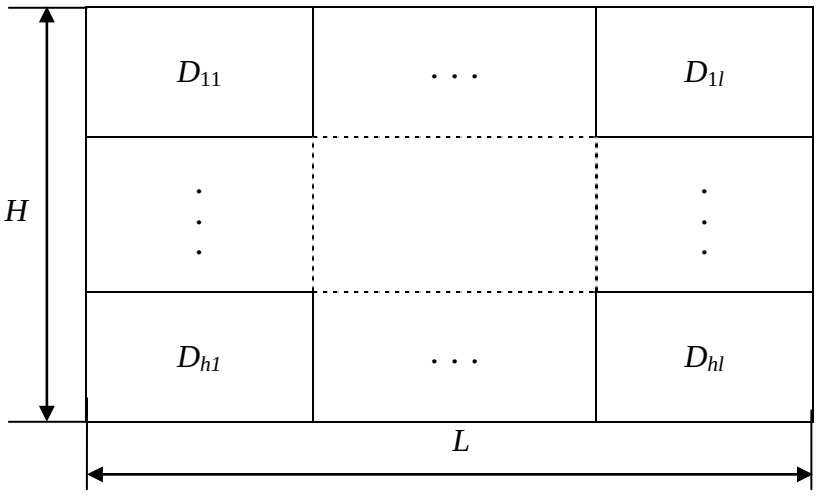


Рис. 4.2. Блочная схема декомпозиции

межпроцессорный обмены имеют место при $H = 2L$. Если $H < 2L$, выгоднее блочная декомпозиция, при $H > 2L$ – ленточная. Ясно, что при другом числе процессоров (подобластей) результаты могут оказаться иными.

Если оказалось, что выгоднее блочная декомпозиция, то следующий важный вопрос – выбор размеров блоков. В большинстве сеточных задач можно полагать, что отношению объема межпроцессорного обмена к объему вычислений в данной подобласти пропорционально отношение длины границ подобластей к их площади. С точки зрения минимизации этого отношения представляется целесообразным подобласти выбирать в форме квадратов или прямоугольников, близких к квадратам. Однако при этом возникает следующая проблема. При разбиении исходной области обработки данных на квадраты одинаковых размеров для фрагментов, расположенных на границах декомпозируемой области, длина границ, сопряженных с соседними фрагментами, а следовательно, и объем передаваемых данных будет меньше. Указанное различие во времени передачи данных может оказывать существенное влияние на эффективность использования процессоров, если скорость передачи данных низкая. Неэффективность использования процессоров более заметна, когда число областей, на которые разбивается изображение, невелико. Повышение эффективности использования процессоров может быть достигнуто увеличением размеров областей, находящихся на границах и в углах изображения. В приложении 1 приведен пример построения соотношений для обоснованного выбора размеров областей в разных частях области данных.

Балансировка загруженности процессоров, безусловно, является важнейшим фактором повышения эффективности системы. Однако при этом должно выполняться допустимое соотношение временных затрат на проведение вычислений на фрагментах и пересылку данных (накладные расходы, в приложении 1 обозначенные как $\Delta_{доп}$). По мере увеличения числа (а значит, уменьшения размеров) фрагментов объем вычислений на каждом фрагменте уменьшается. При этом относительный вес накладных расходов может возрастать, напри-

мер, вследствие большой латентности (связанной с потерями на передачу сообщения нулевой длины) коммуникационной среды.

Можно рекомендовать следующий простой способ построения эффективной параллельной программы, совмещенный с этапом ее отладки. Размеры фрагментов массива исходных данных уменьшают (соответственно увеличивают число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном запуске программы с увеличенным числом процессоров линейного ускорения не происходит, это означает, что накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров. Этот подход обсуждался в работе [7].

Если задача не допускает распараллеливания по данным, т.е. возможен лишь *параллелизм задач*, трудности существенно возрастают. Подход к программированию, основанный на параллелизме задач, подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач, для каждой из которых пишется собственная программа и загружается в «свой» процессор. Эти подзадачи могут взаимодействовать сложным образом, при этом распределение и закрепление подзадач, обеспечивающее равномерную загрузку процессоров, становится не столь очевидным при декомпозиции по данным. Для построения эффективного кода в данном случае необходимо вскрыть более тонкую структуру параллелизма алгоритма. Наиболее подходящим математическим аппаратом в этом случае являются модели в виде графов.

5. Понятия графа алгоритма и графа системы

В качестве моделей алгоритмов решения задач широко используются ациклические ориентированные графы [1–3]. В рамках этой модели множества операций алгоритма и связей между ними описываются двойкой:

$$G = (V, E), \qquad (5.1)$$

где $V = \{1,...,|V|\}$ – множество вершин графа, представляющих операции алгоритма, а E – множество дуг графа, устанавливающих зависимости между операциями, в частности, $E_{i,j} = (i, j)$ означает, что операция j использует результат выполнения операции i. Свойство *ациклично*сти графа алгоритма состоит в том, что в графе не может быть дуг вида $E_{i,i} = (i, i)$.

Если отсутствуют операции условного перехода, а зависимость между операциями связана лишь с передачей информационных данных, приведенная модель является *информационным* графом. Путь максимальной длины в информационном графе называют критическим. Если дугам графа приписать также веса $c_{ij}, \left(i, j = \overline{1, N}\right)$, характеризующие интенсивность информационного обмена между i-й и j-й вершинами (операциями), граф называется *взвешенным направленным графом*.

Взвешенный направленный граф, наряду с информационными связями, может содержать также управляющие связи, отражающие операции условного перехода. Пример построения взвешенного направленного графа, содержащего как информационные, так и управляющие связи, рассмотрен в приложении 2.

Вычислительная система также может быть представлена в виде графа:

$$G_s\left(P_s, E_s\right). \qquad (5.2)$$

Здесь P_s – множество процессоров, E_s – множество дуг, представляющих линии связи между процессорами. Пропускная способность линий связи характеризуется весами дуг графа $m_{ij}, \left(i, j = \overline{1, N}\right), i, j \in E_s$.

Граф вычислительной системы определяется структурой линий коммутации между процессорами ВС (топологией сети передачи данных). Некоторые типовые схемы коммуникации процессоров приведены на рис. 5.1.

Полный граф (completely-connected graph, или clique) – граф, в котором между любой парой процессоров существует линия связи. Такая топология обеспечивает минимальные затраты при передаче данных, однако оказывается сложной при большом количестве процессоров.

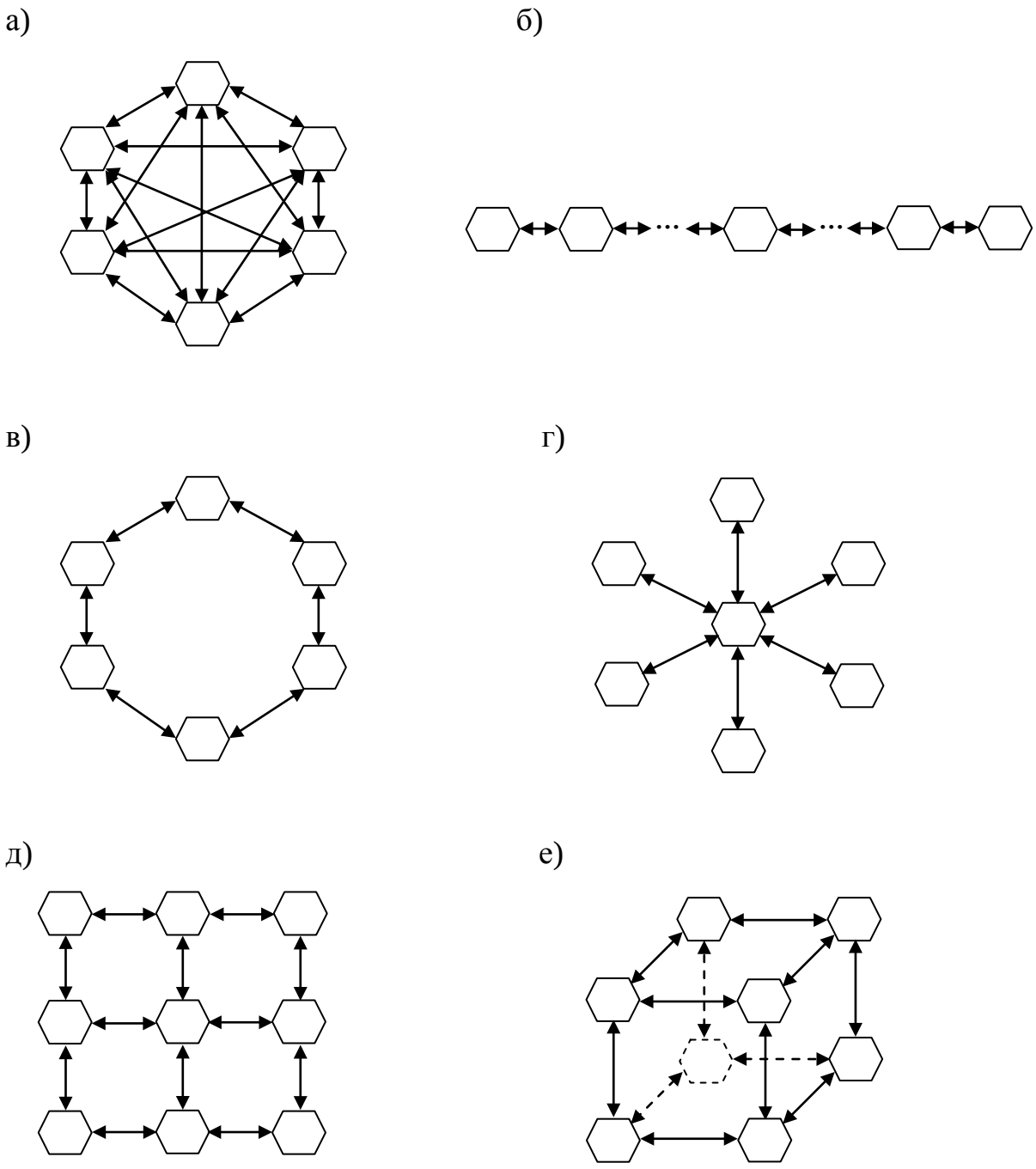


Рис. 5.1. Схемы коммуникации процессоров: а) полный граф; б) линейка; в) кольцо; г) звезда; д) 2-мерная решетка; е) гиперкуб (N = 8 ребра – связи, вершины – процессоры, пронумерованные двоичными цифрами)

Линейка (linear array, или farm) – все процессоры пронумерованы по порядку, и каждый процессор, кроме первого и последнего, имеет линии связи только с двумя соседними. Достоинство – простая реализация, однако класс задач, которым эта схема соответствует, ограничен. Топология является подходящей для конвейерных вычислений.

Кольцо (ring) – получается из линейки процессоров соединением первого и последнего процессоров линейки.

Звезда (star) – все процессоры имеют связь с одним (управляющим) процессором. Топология эффективна для централизованной схемы вычислений.

Решетка (mesh) – граф связей образует двух- или трехмерную сетку. Топология реализуется достаточно просто и может эффективно использоваться при решении сеточных задач, описываемых уравнениями в частных производных.

Гиперкуб (hypercube) – частный случай решетки, когда по каждой размерности сетки имеется только два процессора (при размерности N гиперкуб содержит 2^N процессоров). В N-мерном гиперкубе каждый процессор связан ровно с N соседями, а для сбора данных, например, с $2s$ процессоров, соединенных по схеме гиперкуба, требуется $\log_2 2s$ итераций.

Основные характеристики топологии сети [3]:

- *диаметр* – максимальное (по всем возможным парам) расстояние между двумя процессорами сети, измеряемое по кратчайшему пути, эта величина обычно характеризует максимально время, необходимое для передачи данных между процессорами;
- *связность* (connectivity) – характеризует наличие разных маршрутов передачи данных между процессорами, показатель может быть определен, например, как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на две несвязные области;
- *стоимость* – общее количество линий передачи данных в многопроцессорной вычислительной системе.

В заключение подчеркнем, что архитектура процессоров играет решающую роль. Например, известно, что самый простой способ коммутации процессоров – использование общей шины. Однако в таких системах даже небольшое увеличение числа процессоров, подключаемых к общей шине, делает ее узким местом. Другой классический пример: если решается задача, в которой каждая следующая операция может выполняться лишь после завершения предыдущей,

тогда применение мощного векторного суперкомпьютера ничего не даст. Поэтому при подготовке параллельных алгоритмов и программ необходимо прежде всего ознакомиться со схемой коммутации между процессорами ВС (топологией сети передачи данных) в вычислительной системе, на которой предполагается реализация алгоритма.

6. Анализ параллелизма по графу алгоритма

После того как модель алгоритма представлена в виде графа, можно построить модель параллельных вычислений. Для этого необходимо провести анализ параллелизма алгоритма, т.е. выявить, какие операторы могут выполняться параллельно и независимо. Анализ параллелизма графов алгоритмов, содержащих связи по управлению, обычно сложнее анализа параллелизма информационных графов. Рассмотрим методики анализа параллелизма для обоих указанных типов графов.

Простейшая модель параллельных вычислений для информационного графа может быть построена в виде строгой параллельной формы. Построение этой формы опирается на следующее свойство информационного графа. Все вершины ориентированного информационного ациклического графа с n вершинами можно разметить индексами $1,2,\ldots,s$ таким образом, чтобы для всех дуг, идущих из вершины с индексом i в вершину с индексом j, выполнялось неравенство $i < j$. При этом число s таких индексов не превысит n [2]. Покажем это.

Пометим любое число вершин графа, не имеющих предшествующих, единицей и удалим из графа эти вершины вместе с инцидентными им дугами. Оставшийся граф также ациклический. В нем любое число вершин, не имеющих предшествующих, пометим индексом 2 и удалим их. Продолжим этот процесс до исчерпания графа. Поскольку на каждом шаге отмечается не менее одной вершины, число индексов не превысит число вершин графа.

Нетрудно заметить, что при описанной разметке никакие две вершины с одинаковым индексом не связаны дугой, а минимально число индексов на еди-

ницу больше длины критического пути. Граф, размеченный в соответствии с описанной схемой, называют *строгой параллельной формой графа* [2].

Если на каждом шаге разметки одинаковым индексом отмечаются все вершины графа, не имеющие предшествующих вершин, получающаяся в результате строгая параллельная форма называется *канонической*. Для каждого заданного графа алгоритма каноническая форма *единственна*.

Группа вершин с одинаковыми индексами называется *ярусом*, число вершин в группе – *шириной* яруса, а число ярусов – *высотой* параллельной формы. Параллельная форма минимальной высоты называется *максимальной*, т.к. в каждом ярусе такой формы максимальное число вершин.

Граф алгоритма, размеченный описанным способом, по существу является простейшей моделью параллельного алгоритма. Легко понять, что все операции, отмеченные одинаковыми индексами, могут выполняться одновременно (параллельно). Если для каждого яруса определить время его выполнения, равное времени самой длинной операции, простым суммированием легко подсчитать общее время выполнения алгоритма. Понятно, что минимальное время получится в том случае, если строгая параллельная форма каноническая, а допустимое число используемых процессоров не менее ширины яруса.

Ограничивая число операций, которые могут выполняться параллельно, можно получить отличающиеся строгие параллельные формы. В предельном случае, когда на каждом шаге вычислительного процесса может выполняться только одна операция, т.е. все ярусы имеют ширину, равную 1, будет получена так называемая *линейная* форма, т.е. граф упорядочивается *линейно*. Поэтому модель параллельных вычислений в виде строгой параллельной формы можно построить для любого наперед заданного допустимого числа используемых процессоров.

Описанная модель параллельных вычислений в виде строгой параллельной формы графа может использоваться только для информационного графа. Кроме того, выше мы предположили, что на каждом ярусе доступное число процессо-

ров решающего поля не меньше ширины яруса. В действительности эти предположения часто не выполняются. Если граф содержит соответствующие условным переходам управляющие связи, можно граф разбить на множество информационных графов, для каждого из них провести анализ параллелизма. В частности, необходимо выявить множество взаимно независимых операторов (ВНО) с учетом того, что два участка алгоритма, являющихся ветвями одной операции условного перехода, не могут выполняться при одном запуске программы.

Для графа, содержащего небольшое число операторов и связей, это часто можно сделать путем непосредственного просмотра вариантов реализации. В случае графа с большим числом связей и операторов «ручной» анализ становится затруднительным или даже невозможным. В этом случае для выявления множеств ВНО можно воспользоваться технологией, предложенной в работе [1]. В рамках этой технологии алгоритм представляется в виде так называемой матрицы следования, а затем осуществляется анализ параллелизма графов и определение множества ВНО с использованием формальных матричных преобразований. Пример определения множества ВНО для графа из предыдущего примера, приведенного в приложении 2, рассмотрен в приложении 3.

Из этого примера, в частности, легко заметить, что после того как построена граф-схема алгоритма, все операции с соответствующими матрицами, включая поиск максимально полного множества ВНО, строго формализованы и могут быть реализованы в виде алгоритма. Следовательно, нет никаких препятствий для построения полностью автоматических процедур формирования параллельного алгоритма задачи, после того как алгоритм структурирован, т.е. определены операции и указана последовательность их реализации. Этот этап творческий, основанный на интуиции и опыте.

7. Понятие и построение модели параллельных вычислений

После того как выявлены операторы, которые могут выполняться параллельно, т.е. построена строгая параллельная форма для информационного графа

или установлены все множества ВНО в общем случае графа, содержащего связи по управлению, можно приступить к формированию модели параллельных вычислений. Построение этой модели по существу сводится к составлению расписания выполнения алгоритма, в котором за каждым оператором должен быть закреплен определенный процессор.

Расписание представляют в виде тройки [2]:

$$H_s = \left\{ \left(V_i, P_i, t_i \right) : i = \overline{1,n} \right\}. \qquad (7.1)$$

Здесь P_i – номер процессора, за которым закреплена операция V_i, t_i – время начала выполнения операции, а n – общее число операций. При этом расписание (7.1) совместно с графом вычислительной схемы алгоритма (5.1) рассматривается в качестве модели параллельного алгоритма:

$$A_s(G,H_s), \qquad (7.2)$$

реализуемого с использованием s процессоров.

Использование в этой модели параметра t_i – время начала выполнения операции – создает некоторые неудобства. Для того чтобы всегда реализовывалось выбранное значение этого параметра, начало отсчета времени должно быть связано с моментом запуска этого алгоритма. При анализе показателей эффективности и производительности совместной реализации комплекса алгоритмов это может создавать определенные трудности.

Модернизируем эту модель. В частности, вместо параметра t_i будем использовать множество $P = \{t_i\}, i = \overline{1,n}$, весов вершин, определяющих время выполнения каждого оператора (для упрощения мы полагаем, что вычислительная система однородна и веса зависят только от номера оператора в графе алгоритма).

С учетом сказанного модель параллельных вычислений будем описывать графом

$$G_s = \left(V, P, E, I_s \right), \qquad (7.3)$$

где V – множество вершин графа, представляющих операции алгоритма; $P=\{t_i\}, i=\overline{1,n}$ – множество весов вершин; E – множество дуг графа, устанавливающих зависимости между операциями (для простоты мы не рассматриваем случай, когда дугам также приписаны веса); I_s – множество натуральных чисел $i \in I_s, \quad i=1,2,...,s$, используемых для нумерации s доступных для реализации алгоритма процессоров.

В ходе составления расписания необходимо за каждым процессором (целым числом) закрепить оператор, т.е. каждой операции $v_i \in V$ из множества V поставить в соответствие номер процессора $i \in I_s$. Еще раз подчеркнем, что в этой модели мы отказываемся от использования параметра t_i – время начала выполнения операции. В модели (7.3) при заданном множестве параметров $P=\{t_i\}, i=\overline{1,n}$, определяющих время выполнения каждой операции, множество связей E однозначно определяет время начала выполнения операции, притом так, что к началу каждой операции на любом процессоре все необходимые данные уже вычислены.

Модель параллельных вычислений (7.3) удобно представлять в виде временной диаграммы выполнения операторов. В диаграмме [1] операторы обозначаются прямоугольниками с длиной, равной времени выполнения соответствующего оператора. Номера операторов на диаграмме обозначаются целыми числами в соответствии с их нумерацией в графе алгоритма. Для определенности условимся обозначать их цифрой в кружке, расположенном в центре прямоугольников. Ось ординат диаграммы разбивается на интервалы, каждый из которых соответствует одному из параллельно работающих процессоров. В каждом интервале размещаются только те прямоугольники-операторы, которые закреплены за соответствующим этому интервалу процессором.

Связи между прямоугольниками-операторами обозначаются цифрами слева и справа от номера оператора, указанного в кружочке в центре. В частности, в левой части прямоугольника-оператора записываются номера предшествующих по информационной связи операторов, а в правой части – номера операторов, следующих за данным оператором. Поскольку каждый прямоугольник

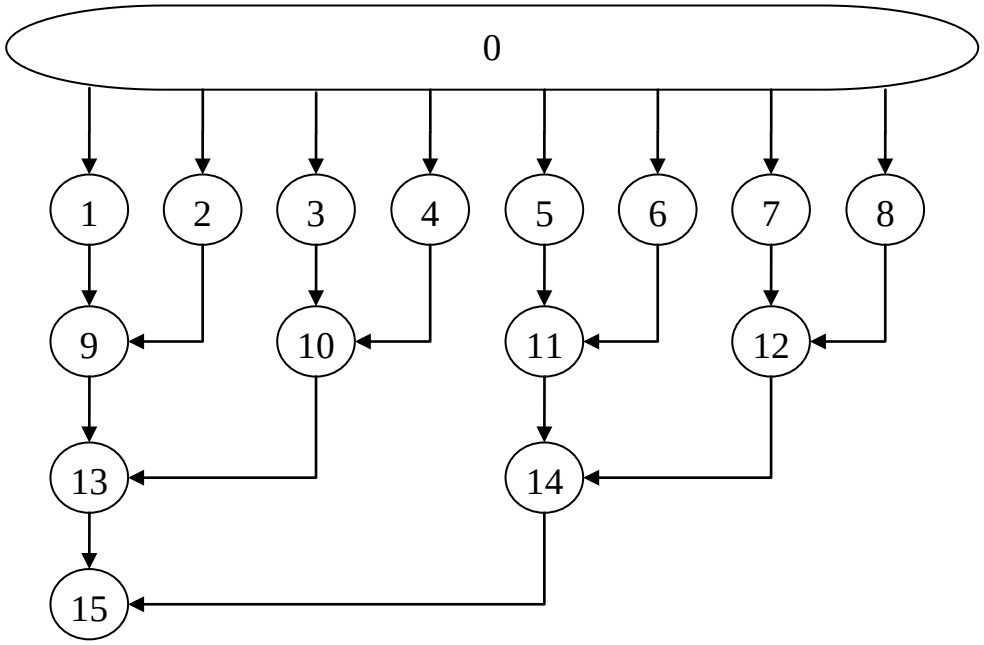
диаграммы размещается в интервале «своего» процессора, описанный способ их нумерации отражает также связь между процессорами.

Описанный способ представления модели параллельных вычислений является исчерпывающим описанием расписания параллельного алгоритма и может быть удобным программисту для написания параллельной программы. При этом информация о номерах связанных операторов может использоваться для оценки объема передаваемых данных как между процессами, так и между процессорами. На рис. 7.1*б* в виде диаграммы, составленной в соответствии с описанными выше правилами нумерации операторов, приведен пример модели параллельных вычислений алгоритма, представленного графом на рис. 7.1*а*.

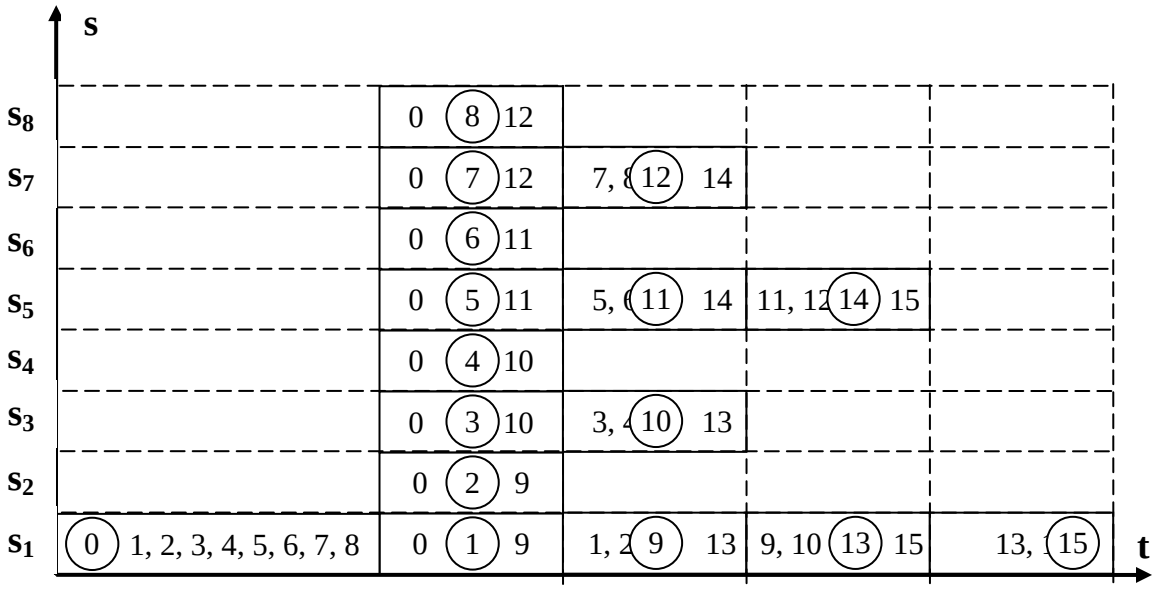
Построение модели параллельных вычислений в случае, когда задача декомпозируется по данным (в рамках инженерного подхода), обычно не представляет трудностей. В этом случае задача сводится лишь к закреплению процессоров за фрагментами, на которые разбита вся область исходных данных. Если при этом после каждой итерации обработки фрагментов требуется обмен данными, необходимо так распределять работы между процессорами, чтобы на каждой итерации длины прямоугольников-операторов были одинаковыми.

В качестве примера построим модель параллельных вычислений в виде временной диаграммы для задачи умножения матрицы на вектор. Выбор этого примера обусловлен тем, что матричные операции являются классическими примерами для демонстрации многих приемов и методов распараллеливания задач. Далее для общности будем полагать, что матрица является плотной, т.е. число нулевых элементов в них мало по сравнению с общим количеством элементов матриц.

При распараллеливании задачи умножения матрицы на вектор обычно используются два типа декомпозиции, показанные на рис. 4.2: ленточное разбиение на полосы и разбиение данных на прямоугольные фрагменты (блочное разделение). Ограничимся рассмотрением ленточного разбиения по строкам, при котором каждому процессору выделяется некоторое количество обычно подряд идущих строк.



а)



б)

Рис. 7.1. Граф (а) и временная диаграмма (б) алгоритма умножения матрицы на вектор при разделении матрицы по строкам

На рис. 7.1*а* приведен граф алгоритма параллельного умножения матрицы на вектор на 8 процессорах при ленточном разбиении по строкам. Здесь цифрой 0 обозначена операция распределения данных (разделения матрицы на ленточные фрагменты и рассылка этих фрагментов и умножаемого вектора процессорам); цифрами 1–8 обозначены операции умножения выделенных каждому процессору полос на вектор. В результате выполнения этих операций на каж-

дом процессоре будет получена 1/8 часть искомого вектора. Цифрами 9–15 обозначены операции «сборки». Сборка осуществляется за три шага. После первого шага (выполнения операций 9–12) на 4 процессорах окажется по 1/4 части искомого вектора. На следующем шаге (операции 13–14) на двух процессорах будет сформировано по 1/2 части искомого вектора. Наконец, на завершающем 3-м шаге в результате выполнения операции 15 на процессоре № 1 будет получен результирующий вектор.

Соответствующая этому графу временная диаграмма параллельных вычислений приведена на рис. 7.1*б*. Если подготовлены фрагменты программы, реализующие фигурирующие на графе алгоритма операции, то по этой диаграмме нетрудно «собрать» параллельную программу решения всей задачи. Мы продолжим этот пример в главе 9, в частности, приведем MPI-программу, реализующую эту модель параллельных вычислений.

В случае, когда имеет место параллелизм задач, составление расписания обычно оказывается серьезной проблемой. На рис. 7.2*а* приведен простой пример графа алгоритма такого типа, а на рис. 7.2*б,в,г* – возможные варианты модели параллельных вычислений. Как видно из рис. 7.2*г,* ценой увеличения времени решения задачи на один временной интервал число используемых процессоров может быть уменьшено до двух. Обоснованный выбор подходящего варианта модели параллельных вычислений даже в рассматриваемом простом случае оказывается нетривиальным.

При построении модели параллельных вычислений очень важно сформулировать цель. Например, если мы хотим построить реализацию некоторого информационного графа с максимально возможным быстродействием, то ясно, что вычислительный процесс должен быть организован в соответствии с канонической строгой параллельной формой. При этом максимальное число одновременно работающих процессоров должно быть равным максимальной ширине яруса. Однако если число доступных процессоров меньше максимальной ширины яруса или число ярусов максимальной ширины мало, а их ширина велика, то целесообразно минимизировать число используемых процессоров, чтобы исключить простаивание большей их части.

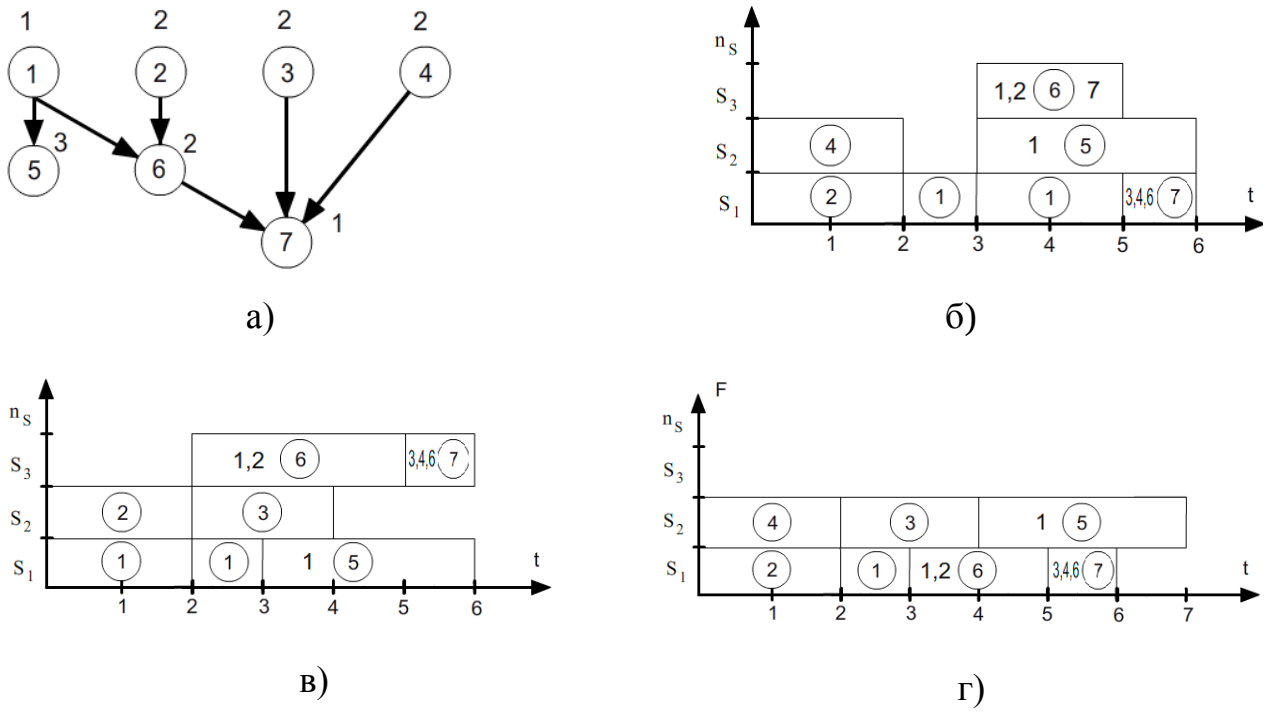


Рис. 7.2. Примеры построения временных диаграмм:

а) граф; б)-г) варианты временных диаграмм

Конечно, с точки зрения повышения быстродействия реализации алгоритма всегда необходимо стремиться максимально «подогнать» граф вычислительной системы к графу алгоритма. Однако при этом, как мы заметили выше, должны выполняться также другие целевые установки и ограничения. Решение этой задачи является содержанием центральной проблемы при планировании вычислительных ресурсов – *проблемы отображения* параллельного алгоритма и соответствующей ему программы на архитектуру многопроцессорной вычислительной системы.

Общая постановка задачи следующая. Задается критерий оптимальности отображения $\varphi : V_p \to P_s$ графа параллельной задачи $G_p(A_p, E_p)$ на структуру вычислительной системы, заданной графом (5.2) – $G_s(P_s, E_s)$:

$$Q(\varphi) = Q\{\varphi : V_p \to P_s\}. \qquad (7.4)$$

Решение задачи отображения в общем случае, когда граф алгоритма содержит как управляющие, так и информационные связи, – крайне трудная про-

блема. Задача упрощается путем декомпозиции графа на блоки, содержащие графы алгоритмов, при одной реализации которых выполняются все операторы. Как отмечалось выше, такие алгоритмы, в которых отсутствуют связи по управлению, представляются информационными графами.

Такими укрупненными информационными графами можно описывать также алгоритмы, в которых операторами (возможно, с изменяющимися от реализации к реализации весами) являются крупные взаимодействующие подзадачи, даже если эти подзадачи содержат логические операторы. Если выделить такие крупные позадачи не удается и решается задача распараллеливания алгоритма с логическими операторами, все равно необходимо рассмотреть все возможные варианты, в которых все операторы обязаны выполняться при одной реализации. Каждый отдельно взятый вариант при этом представляется информационным графом.

Для информационного графа в рамках проблемы отображения (7.4) обычно решается одна из двух задач:

Задача 1. Для данного алгоритма, которому соответствует информационный граф *G* со скалярными весами вершин, и для времени *T*, отведенного для его выполнения, найти наименьшее необходимое число *s* процессоров, входящих в состав однородной вычислительной системы (ВС), и план выполнения операторов на них.

Задача 2. Для заданного алгоритма, которому соответствует информационный граф *G* со скалярными весами вершин, найти минимальное время *T* и план реализации этого алгоритма на данной однородной ВС, в состав которой входят *s* процессоров.

Решение задачи 1 заключается в определении минимального числа *s*, для которого можно построить преобразованный граф $G_s = (V, P, E, I_s)$, объединив вершины, соответствующие операторам каждого полного множества ВНО, содержащего $r > s$ операторов $r - s$ ориентированными дугами в *s* путей, не со-

держащих общих вершин. При этом длина критического пути в графе G_s не должна превышать значение T.

Описанная выше задача определения минимально необходимого числа процессоров может быть решена методом направленного перебора (ветвей и границ). В данном случае ветвление определяется различными способами введения дополнительных связей в каждое полное множество ВНО, число операторов в котором превышает s, или его подмножество. Границы определяются допустимыми изменениями длины критического пути в графе после введения дополнительных связей.

Алгоритм нахождения графа G_s в этом случае может строиться следующим образом. Сначала выбирается некоторое начальное расписание для определенного числа процессоров. Затем, задавая другой вариант дополнительных связей и закрепления операторов за процессорами, допускаемых ограничением на T (длину критического пути), пытаются уменьшить число процессоров s. Если это не удается, вводят другую комбинацию связей и закрепления процесоров. При переборе всех комбинаций связей число процессоров увеличивают на единицу и начинают процесс сначала.

Подчеркнем, что построение графа $G_s = (V, P, E, I_s)$ по описанной схеме является достаточным для решения задачи 1. Действительно, предположим внутри каждого полного множества ВНО с числом операторов $r > s$ в соответствии с описанной схемой введено $r - s$ связей и s – наименьшее из целых чисел, для которых это возможно. Тогда в сформированном графе G_s каждое полное множество ВНО содержит не более s операторов. Но это означает, что минимальное число процессоров, способных выполнить данный алгоритм за время T, не может быть больше s. С другой стороны, допущение о том, что число процессоров меньше s, противоречит результату работы алгоритма, в соответствии с которым s – наименьшее из целых чисел, для которого возможно введение $r - s$ связей.

Для закрепления процессоров в ходе решения задачи необходимо просматривать значения ранних сроков начала выполнения операторов в порядке их неубывания и закреплять за процессорами, которые в эти моменты свободны. Определение ранних сроков выполнения операторов не представляет трудностей, т.к. известны веса вершин – $P = \{t_i\}, i = \overline{1,n}$, определяющие время выполнения каждого оператора и множество дуг графа – *E*, устанавливающих зависимости между операциями. Для этого достаточно «пройти» по всем дугам от начала алгоритма до данного оператора и просуммировать встречающиеся на этом пути веса вершин.

В основе алгоритма решения задачи 2 лежит та же идея, что и при решении задачи 1. Для того чтобы *T* было минимальным временем выполнения алгоритма, представленного информационным графом $G = (V, P, E)$ на *s* процессорах, достаточно построить граф $G_s = (V, P, E, I_s)$, для которого длина критического пути минимальна среди всех графов, полученных из заданного путем объединения вершин, соответствующих операторам каждого полного множества ВНО, содержащего $r > s$ операторов, $r - s$ ориентированными дугами в s путей, не содержащих общих вершин.

Действительно, пусть T – минимальное время, для которого удалось построить такой граф $G_s = (V, P, E, I_s)$ и $T = T_{кр}^{'}$. Каждое полное множество ВНО в графе G_s содержит не более *s* операторов. Следовательно, алгоритм, представленный графом G_s, может быть выполнен за время $T_{кр}^{'} = T$.

Для решения этой задачи может использоваться такая же схема, как и при решении задачи 1. Сначала выбирается некоторое начальное расписание, при реализации которого на *s* процессорах обеспечивается $T = T_0$. Затем, задавая другой вариант дополнительных связей и закрепления операторов за процессорами, допускаемых ограничением на число прооцессоров, проверяется возможность уменьшения времени реализации по сравнению с T_0. Если это не удается, вводят другую комбинацию связей и закрепления процесоров и т.д.

Решение обоих задач существенно усложняется, если вычислительная система неоднородная и веса $P = \{t_{i,j}\}$, $i = \overline{1,n}$, $j = \overline{1,s}$, вершин, определяющих время выполнения каждого оператора, зависят не только от номера оператора в графе алгоритма, но и от номера процессора, за которым он закреплен на данном этапе решения задачи. Эту задачу в рамках настоящего учебного пособия мы не рассматриваем.

8. Построение оценок производительности и эффективности параллельных алгоритмов

Для построения оценок производительности и эффективности мы будем использовать следующую модель вычислительной системы [2]. Система состоит из набора *простых функциональных устройств* (ФУ), не имеющих *собственной памяти*, т. е. никакая последующая операция не может начаться раньше, чем предыдущая, а результат предыдущего срабатывания ФУ может сохраняться в нем только до момента очередного срабатывания ФУ.

Обозначим τ – время выполнения одной операции (*стоимость операции*), а T – время выполнения работы (*стоимость работы*). За это время может быть выполнено приблизительно T/τ операций (в действительности следует брать только целую часть результата деления). *Загруженностью устройства – pE* называют отношение стоимости реально выполненной работы к максимально возможной стоимости.

Максимальный объем работ, который может быть выполнен системой, оценивается *пиковой производительностью*, определяемой как максимальное количество операций, которое может быть выполнено системой за единицу времени при отсутствии потерь времени на связи между ФУ. Пиковая производительность определяется как количество операций с плавающей точкой (простых ФУ), выполняемых за один такт, умноженное на частоту работы процессора и на число процессоров. Единица измерения производительности – Flops (одна вещественная операция в секунду).

Показатель эффективности одного процессора – количество операций, запускаемых за один такт процессора – IPC (instructions per sycle). *Реальная производительность* вычислительной системы – это количество операций, реально выполняемых в среднем в единицу времени. Отношение реальной производительности к пиковой характеризует *эффективность* реализации задачи на данном конкретном компьютере. Превышение пиковой производительности над реальной характеризует, насколько данная архитектура приспособлена к решению конкретной задачи.

Если s устройств системы имеют пиковые производительности $\pi_1,\ldots,\pi_s$ и работают с загруженностями $p_1, \ldots, p_s$, то реальная производительность системы выражается формулой [2]

$$r = p\pi, \tag{8.1}$$

где $\pi = \pi_1 + \ldots + \pi_s$, а p – *загруженность системы*, определяемая как

$$p = \sum_{i=1}^{s} \alpha_i p_i, \quad \alpha_i = \frac{\pi_i}{\sum_{j=1}^{s} \pi_j}. \tag{8.2}$$

Из (8.1) видно, что для достижения наибольшей реальной производительности системы при фиксированном числе устройств необходимо обеспечить наиболее полную ее загруженность. Дальнейшее повышение производительности достигается увеличением числа устройств.

Ускорение реализации алгоритма на вычислительной системе из s устройств определяется как [2]

$$R_s = \frac{r}{\pi_s}, \tag{8.3}$$

где π_s – пиковая производительность самого быстродействующего устройства системы. Это означает, что наибольшее ускорение – s системы из s устройств может достигаться только в случае, когда все устройства системы имеют одинаковые пиковые производительности и полностью загружены.

Реальное ускорение для однородных вычислительных систем, имеющих одинаковую производительность устройств, часто определяют также как отно-

шение времени решения задачи на одном процессоре – T_1 ко времени T_s решения той же задачи на системе из s таких же процессоров:

$$R = \frac{T_1}{T_s}. \tag{8.4}$$

Это соотношение можно получить также из (8.3) с учетом (8.1), т.к. при одинаковой производительности устройств $p = T_1 / T_s \cdot s$, а $\pi = \pi_s \cdot s$.

Отношение *реального ускорения* к числу используемых процессоров s:

$$E_s = \frac{R}{s} = \frac{T_1}{T_s \cdot s} \tag{8.5}$$

называют *эффективностью* системы. Второе равенство в (8.5) показывает, что при одинаковой производительности устройств эффективность системы совпадает со значением загруженности системы. Наилучшие показатели ускорения и эффективности – соответственно $R = s$, $E_s = 1$.

Для анализа производительности вычислительных систем, в которых имеют место направленные связи между устройствами, используют модель ориентированного *графа* [2]. В частности, с использованием этой модели в приложении 4 показано, что для системы из s устройств с пиковыми производительностями $\pi_1,\ldots,\pi_s$, описываемой связным графом (5.2), максимальная производительность $r_{\max}$ определяется как

$$r_{\max} = s \min_{1 \le i \le s} \pi_i .$$

Отсюда вытекает 1-й *закон Амдала*: производительность вычислительной системы, состоящей из связанных между собой устройств, определяется самым непроизводительным устройством.

Для количественной оценки производительности многопроцессорных вычислительных систем используется 2-й *закон Амдала* [2]. В соответствии с этим законом максимально возможное ускорение для системы, состоящей из s одинаковых устройств, определяется как

$$R = \frac{s}{\beta \cdot s + (1 - \beta)}, \tag{8.6}$$

где $\beta = n/N$, N – общее число операций алгоритма, а n – число операций, которые могут выполняться только последовательно. Доказательство соотношения (8.6) приведено в приложении 5. Формула Амдала используется для прогноза достижимого ускорения на этапе анализа параллелизма задачи. Например, если по описанию алгоритма установлено, что половина операций не поддается распараллеливанию, максимально достижимое ускорение в случае использования 2 процессоров в соответствии с (8.6) составит около 1,33, для 10 процессоров – менее 1,82, а для 100 процессоров – около 1,98.

Наряду с ускорением и эффективностью, важным показателем является также масштабируемость вычислительной системы. Параллельный алгоритм называют масштабируемым (scalable), если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении эффективности использования процессоров.

Для характеристики свойств масштабируемости оценивают накладные расходы (время T_0) на организациию взаимодействия процессоров, синхронизацию параллельных вычислений и т.п.:

$$T_0 = sT_s - T_1, \tag{8.7}$$

где T_s, T_1 – те же, что и в (8.4). Используя эти обозначения, соотношения для времени параллельного решения задачи и соответствующего ускорения можно представить в виде

$$T_s = \frac{T_1 + T_0}{s}, \tag{8.8}$$

$$R_s = \frac{T_1}{T_s} = \frac{sT_1}{T_1 + T_0}. \tag{8.9}$$

Соответственно эффективность использования s процессоров

$$E_s = \frac{R_s}{s} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0/T_1}. \tag{8.10}$$

Из (8.10) следует, что если время решения последовательной задачи фиксировано (T_1 = const), то при росте числа процессоров эффективность может убывать лишь за счет роста накладных расходов T_0.

Если число процессоров фиксировано, эффективность их использования, как правило, растет при повышении времени (сложности) решаемой задачи T_1. Связано это с тем, что при росте сложности задачи накладные расходы T_0 обычно растут медленнее, чем объем вычислений T_1. Для характеристики свойства сохранения эффективности при увеличении числа процессоров и повышении сложности решаемых задач строят так называемую функцию изоэффективности.

Пусть задан желаемый уровень эффективности выполняемых вычислений:

E_s = const.

Из выражения для эффективности (8.20) можно записать

$$\frac{T_0}{T_1} = \frac{1 - E_s}{p_s}.$$

Или

$$T_1 = KT_0, \quad \text{где} \qquad K = \frac{E_s}{1 - E_s}.$$

Из последнего равенства видно, что эффективность характеризуется коэффициентом K. Следовательно, если построить функцию вида

$$N = F(K, s),$$

то для заданного относительного уровня эффективности K каждому числу процессоров s можно поставить в соответствие требуемый уровень сложности задачи в виде общего числа операций – N и наоборот. При рассмотрении конкретных вычислительных алгоритмов построение функции изоэффективности помогает выявить направление совершенствования параллельного алгоритма.

Для построения этих функций удобно использовать закон *Густавсона–Барсиса*:

$$R = \frac{T_1}{T_s} = g + (1-g)s = s + (1-s)g\,. \qquad (8.11)$$

где $g = \dfrac{\tau_n}{\tau_n + \tau_{N-n}/s}$, (8.12)

а τ_n и τ_{N-n} – время, необходимое для выполнения последовательной и параллельной частей соответственно (доказательство соотношений (8.11) приведено в приложении 6).

Эффективность использования s процессоров в соответствии с законом Густавсона–Барсиса (8.11) выражается в виде

$$E_s = \frac{R}{s} = 1 + \frac{(1-s)}{s} g\,.$$

При заданном фиксированном E_s = const с использованием этого равенства можно построить аналитическое соотношение для функции изоэффективности в следующем виде:

$$g = F(E_s, s),$$

где g – указанная выше в (8.12) доля времени на проведение последовательных расчетов в выполняемых параллельных вычислениях.

Приведенные соотношения закона Амдала, Густавсона–Барсиса и функции изоэффективности дают границы для достижимой производительности и масштабируемости. Для прогноза достижимых ускорения и эффективности можно также провести непосредственный детальный анализ вычислительных затрат и накладных расходов с использованием конкретных технических характеристик вычислительной системы. Пример построения соотношений для оценки ускорения и эффективности реализации параллельного алгоритма умножения матрицы на вектор с разбиением матрицы по строкам приведен в приложении 7. Для построения соотношений использовалась временная диаграмма алгоритма, приведенная на рис. 7.1*б*.

В заключение этого раздела, посвященного анализу производительности, заметим, что для начинающего пользователя может показаться удивительным тот факт, что, запустив программу на компьютере с огромной пиковой произ-

водительностью, можно не получить ускорение или даже замедление счета по сравнению с обычным персональным компьютером. Для правильной оценки ситуации в этом случае прежде всего необходимо сопоставить граф алгоритма и граф вычислительной системы. Наряду с общими, влияющими на производительность факторами, которые рассматривались выше, часто недооценивается влияние кэш-памяти. Эффективность кэш-памяти существенно выше, если в задаче большой объем *локальных вычислений* и *локального использования данных*. В моменты времени, когда объем локальных данных превышает размеры кэш-памяти, наблюдается резкое падение производительности вычислительной системы.

При решении задач на кластере одним из основных факторов являются также *коммутационные сети.* Они определяют накладные расходы – время задержки передачи сообщения. Это время зависит от латентности (начальной задержки при посылке сообщений) и длины передаваемого сообщения. Для правильной трактовки временных затрат надо сопоставлять характер и множество обменов по графу алгоритма с характеристиками используемой сети. Конечно, на производительность параллельного компьютера огромное влияние оказывают также операционная система, драйвера сетевых устройств, программы, обеспечивающие сетевой интерфейс нижнего уровня, компиляторы и др. Однако этими факторами программист уже не может управлять в полной мере, поэтому мы их не обсуждаем. Достаточно подробное рассмотрение этих вопросов можно найти в учебных пособиях, посвященных параллельному программированию [3, 7].

9. Подготовка и запуск параллельных программ

Цель настоящей главы – дать начальные сведения по подготовке, отладке и запуску параллельных программ на кластере. При этом имея в виду скорейшее освоение технологии кластерных вычислений, мы ограничимся рассмотрением минимального набора программных средств для подготовки параллельных приложений. Для быстрого знакомства с этими инструментами мы рассмотрим

их применение в простой программе ввода-вывода данных, а также продолжим рассмотренный в главе 7 пример умножения матрицы на вектор с ленточной декомпозицией по строкам.

При реализации параллельного алгоритма на многопроцессорной вычислительной системе (например, кластерной архитектуры) запускаются программы, написанные на языке последовательного программирования Си или Фортран. Каждая такая программа выполняет некоторую операцию обработки «своей» порции данных. Взаимодействие этих программ обеспечивает библиотека **MPI** (Message Passing Interface – интерфейс передачи сообщений) путем вызова функций синхронизации, посылки и получения сообщений. Работа с библиотекой **MPI** несколько различается при программировании на языках Си и Фортран. Здесь излагается вариант библиотеки при программировании на Си.

MPI-программа запускается одновременно на параллельно работающих процессорах. Каждая копия программы реализует отдельный процесс. Процессы изолированы друг от друга в том смысле, что у них разные области кода, стека и данных. Процессы обмениваются друг с другом данными в виде сообщений. Сообщения обозначаются идентификаторами. Каждый процесс может получать информацию о количестве одновременно запущенных процессов и свой номер. Процессы могут объединяться в группы.

Важной особенностью MPI является понятие так называемой области связи (*communication domains*). В программе для описания области связи используются коммуникаторы. При запуске программы для описания стартовой области связи библиотека автоматически создает коммуникатор с использованием идентификатора

MPI_COMM_WORLD.

В области связи размещаются все процессы [7], а коммуникатор ограничивает многие функции MPI, прикрепляя их к этой области связи. Если для одной области связи используется несколько коммуникаторов, программа работает с ними как с несколькими разными областями.

Все идентификаторы начинаются с префикса «**MPI_**». Поэтому нельзя использовать собственные идентификаторы, начинающиеся с этого префикса, а также с префиксов

«MPID_», «MPIR_» *и* «PMPI_»,

которые применяются в служебных целях. Имена констант (и неизменяемых пользователем переменных) записываются полностью заглавными буквами:

MPI_COMM_WORLD,

MPI_FLOAT.

В именах функций первая за префиксом буква заглавная, остальные маленькие:

MPI_Send, MPI_Comm_size.

Подчеркнем, что состояние регистра символов при указании имен функций (процедур) и именованных констант при программировании на языке Си существенен (это не имеет значения только при программировании на Фортране).

Определение всех именованных констант, прототипов функций и определение типов выполняется в языке Си подключением файла

mpi.h.

В операциях пересылки и преобразования данных указываются собственные типы **MPI**. Это обеспечивает переносимость программ между различными компиляторами и платформами, а также возможность автоматического преобразования типов данных при пересылках, когда программа реализуется на неоднородной вычислительной системе.

В табл. 9.1, заимствованной из [12], приведен список типов библиотеки **MPI** и соответствующих типов языка Си. Здесь тип

MPI_BYTE

используется для передачи двоичной информации без ее преобразования. Тип

MPI_PACKED

используется для передачи в одном сообщении предварительно упакованных разнотипных данных. Программисту предоставляются также средства создания собственных типов на базе стандартных.

Таблица 9.1. Соответствие между **MPI**-типами и типами языка Си

Тип MPI	Тип языка Си
MPI_CHAR	**signed char**
MPI_SHORT	**signed short int**
MPI_INT	**signed int**
MPI_LONG	**signed long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

Рассмотрим 4 функции **MPI** на примере простой программы [7], в результате выполнения которой каждый запускаемый процесс дожен выдать следующее сообщение:

«Hello world from process i of n»

Здесь **i** – номер процесса, а **n** – количество процессов. Файл с текстом этой программы назовем «**helloworld.c**». Текст программы на языке Си имеет вид:

```
#include
#include "mpi.h"

int main( int argc; char **argv )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

В приведенном тексте программы использованы следующие 4 функции MPI: **MPI_Init**, **MPI_Comm_size**, **MPI_Comm_rank**, **MPI_Finalize**. MPI-программа начинается с вызова функции инициализации MPI:

MPI_Init.

В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы, и создается область связи, описываемая коммуникатором

MPI_COMM_WORLD.

Процессы в группе упорядочены и пронумерованы от 0 до **groupsize-1**, где **groupsize** равно числу процессов в группе. В данном случае величина **groupsize** равна числу процессоров, выделенных задаче. При инициализации каждому процессу передаются аргументы функции

main,

полученные из командной строки.

Функция определения числа процессов в области связи **MPI_Comm_size**:

```
int MPI_Comm_size(MPI_Comm comm, int *size).
```

Здесь коммуникатор **comm** является входным параметром, а **size** – выходной параметр, указывающий число процессов в области связи коммуникатора **comm**, т.е. эта функция возвращает количество процессов в области связи коммуникатора comm. Если явным образом не создаются группы и связанные с ними коммуникаторы, то значениями параметра **COMM** являются **MPI_COMM_WORLD** и **MPI_COMM_SELF**, которые создаются автоматически при инициализации MPI.

Функция определения номера процесса **MPI_Comm_rank**:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank).
```

Здесь коммуникатор **comm** также является входным параметром, а **rank** – выходной параметр, указывающий номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..**size-1** (значение **size** определяется с помощью предыдущей функции).

Функция завершения MPI программ – **MPI_Finalize**:

```
int MPI_Finalize(void).
```

Функция закрывает все MPI-процессы и ликвидирует все области связи.

Для запуска MPI-программы используется команда

mpirun .

Это специальный командный файл – скрипт, в котором указывается количество запускаемых процессов и имя файла исполняемого модуля (программы). Можно также задать конфигурацию (список компьютеров), на которой запускается программа. Количество процессов не обязательно равно количеству процессо-

ров. Возможен запуск MPI-программы с количеством процессов большим, чем число доступных процессоров, при этом несколько процессов должны одновременно выполняться на одном процессоре. Выполнение программы с указанием запуска нескольких процессов на однопроцессорном компьютере позволяет производить первоначальную отладку параллельного приложения. Рассмотрим основные соглашения по использованию библиотеки MPI.

Откомпилируем программу **helloworld.c**. Для этого сформируем в командной строке команду

```
% mpicc -o helloworld helloworld.c
```

Запустим программу на 4 процессорах, задав в той же строке команду:

```
% mpirun -np 4 helloworld
```

Поскольку порядок появления сообщений от различных процессов не определен, в результате выполнения программы мы можем получить следующее сообщение:

```
    Hello world from process 0 of 4
    Hello world from process 3 of 4
    Hello world from process 1 of 4
    Hello world from process 2 of 4
%
```

Если появляется какое-либо сообщение об ошибке, например:

```
    mpicc: Command not found.
```

то скорее всего необходимо в переменной окружения **PATH** указать каталог, где находятся исполняемые файлы библиотеки MPI. Например,

```
    setenv PATH /usr/local/mpi/bin:$PATH
    rehash
```

Как видно из этого примера, рассмотренные четыре функции MPI используются в параллельной программе всегда, даже если программа содержит лишь ввод и вывод данных. Теперь рассмотрим еще две функции MPI, осуществляющие собственно распараллеливание процессов:

```
    MPI_Send, MPI_Recv
```

Эти функции мы рассмотрим на примере программы (далее называемой **Multiply.c**) параллельного умножения матрицы на вектор при ленточном разбиении матрицы по строкам. Модель параллельных вычислений в виде временной диаграммы этого алгоритма была нами рассмотрена в главе 7 (рис. 7.1*б*). Текст Си-программы **Multiply.c** с комментариями приведен в приложении 8.

Рассмотрим более детально функции передачи (**MPI_Send)** и приема (**MPI_Recv**) сообщений. Функция

```
int MPI_Send(void*buf,int count,MPI_Datatype datatype,
int dest,int tag,MPI_Comm comm)
```

выполняет посылку **count** элементов типа datatype сообщения с идентификатором **tag** процессу **dest** в области связи коммуникатора **comm**. Переменная **buf** – это, как правило, массив или скалярная переменная. В последнем случае значение **count = 1**.

Функция

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Здесь входной параметр **buf** – адрес начала расположения принимаемого сообщения, выходные параметры: **count** – максимальное число принимаемых элементов; **datatype** – тип элементов принимаемого сообщения; **source** – номер процесса-отправителя; **tag** – идентификатор сообщения; **comm** – коммуникатор области связи, а также выходной модифицируемый параметр **status** – атрибуты принятого сообщения. Функция выполняет прием **count** элементов типа **datatype** сообщения с идентификатором **tag** от процесса **source** в области связи коммуникатора **comm**.

В функции **MPI_Recv** при указании номера процесса-отправителя возможно использование специального параметра **MPI_ANY_SOURCE** («принимай от кого угодно»), а в качестве идентификатора получаемого сообщения – **MPI_ANY_TAG** («принимай что угодно»). Это так называемые параметры-джокеры, MPI резервирует для них отрицательные целые числа, в то время как реальные идентификаторы процессов и сообщений лежат всегда в диапазоне от

0 до 32767. Пользоваться джокерами следует с осторожностью, потому что по ошибке таким вызовом **MPI_Recv** может быть захвачено сообщение, которое должно приниматься в другой части процесса-получателя.

Если логика программы достаточно сложна, использовать джокеры можно только в функциях, проверяющих наличие сообщения для процесса (**MPI_Probe** и **MPI_Iprobe**), чтобы перед фактическим приемом узнать тип и количество данных в поступившем сообщении. Несмотря на то, что мы хотим получить «что угодно», тип принимаемых данных в функции **MPI_Recv** должен быть указан явно, а он может быть разным в сообщениях с разными идентификаторами.

В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов – передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. В дополнение к стандартному режиму возможно использование синхронной, буферизованной или согласованной передачи как с блокировкой, так и без блокировки. Вместе с тем с помощью только пары операций **MPI_Send/MPI_Recv** можно реализовать практически любой параллельный алгоритм.

В настоящем пособии мы намеренно ограничились изучением лишь 6 функций MPI, достаточных для написания простейших параллельных программ. Это отвечает нашей цели: помочь пользователям в максимально короткие сроки освоить подготовку и реализацию простейших параллельных MPI-приложений на кластере. Пользователи, освоившие работу с описанными в настоящем пособии шестью функциями MPI, могут попытаться построить более эффективный параллельный код с использованием так называемых коллективных функций MPI. Подробное описание коллективных функций и примеры параллельных программ, в которых они используются, можно найти в учебном пособии [5].

В заключение приведем краткую инструкцию по компиляции и запуску MPI-программ. Процесс запуска MPI-программы включает в себя следующие шаги: выбор реализации MPI, компиляцию MPI-программы и запуск MPI-программы.

Для выбора MPI среды выполните следующие команды:

```
[user@mgt1 ~] $ module avail
/etc/modulefiles
impi/3  impi/4   openmpi
[user@mgt1 ~] $ module load impi/4
[user@mgt1 ~] $
```

В результате выполнения этих команд модуль **impi/4** загружает переменные среды для работы с Intel MPI-библиотекой версии 4. Загруженные переменные сохраняются в течение текущей сессии. После выбора реализации MPI можно использовать программы для компиляции и запуска параллельных приложений (**mpicc, mpicxx, mpif77, mpif90** и т. д.).

Для компиляции С-программы используется команда

mpicc – o <имя файла с объектным кодом> <имя файла с исходным кодом>

В данном случае такая команда имеет вид:

[user@mgt1 ~] $ **mpicxx – o Multiply.mpi Multiply.c**

Завершающий этап – запуск программы. Для постановки задания в очередь и передачи всех переменных среды на вычислительные ноды (узлы) используется команда **qsub** с ключом –**V**.

Например, для командного файла задания **run.pbs:**

[user@mgt1 ~] $ **qsub –V run.pbs**

Помимо специфических инструкций, отличающихся для каждого суперкомпьютера, в файле задания должна присутствовать строка запуска программы:

```
mpirun -np 4 Multiply.mpi a.txt x.txt y.txt
```

Для того, чтобы программа успешно выполнилась и выдала верный результат, необходимо, чтобы в директории, из которой запускается программа, присутствовали два файла: **a.txt** и **x.txt**. Файлы должны иметь следующую структуру:

a.txt:

<Число строк>

<Число столбцов>

<Матрица, в которой значения элементов разделены пробелами, а строки – знаком переноса строки>

`x.txt:`

<Вектор, на который необходимо умножить матрицу>

Результат выдается в файл **y.txt**, в котором будет содержаться вектор значений, полученных после умножения, разделенных пробелом.

Для получения информации о статусе запущенной задачи (списка очередей и их параметры; списка задач с расширенной информацией; полной информации о задаче; информации, на каких узлах запущена задача) можно использовать команду **`qstat`**.

Приложение 1. Построение соотношений для обоснованного выбора размеров фрагментов области данных с учетом их локализации

Рассмотрим задачу разбиения исходной области данных на квадратные блоки с учетом локализации подобластей, при котором время работы всех процессоров с учетом пересылок максимально сбалансировано. Для простоты рассмотрим случай, когда исходная область квадратная: $X \times X$, где X – число отсчетов одной стороны области. Будем полагать, что для данного вычислительного алгоритма, реализуемого на заданной вычислительной системе, известны: τ_p – время расчета при обработке одного отсчета (точки) области и τ_n – среднее время, затрачиваемое на передачу информации, необходимой для одной точки области.

Область данных разобьем на прямоугольные фрагменты. Пусть x – сторона фрагмента, численно равная числу точек. Потребуем, чтобы величина x удовлетворяла неравенству

$$\Delta_{доп} \le \left(4 \cdot x \cdot \tau_n\right) \Big/ \left(x^2 \tau_p\right) = \le \left(4 \cdot \delta\right) \Big/ x, \qquad (П1.1)$$

где $\delta = \tau_n / \tau_p$ – отношение отрезков времени, необходимых для пересылки данных ко времени обработки в расчете на одну точку области, а $\Delta_{доп}$ – допустимая величина отношения времени пересылок ко времени обработки внутренней области, задаваемая из условия эффективной загрузки процессоров.

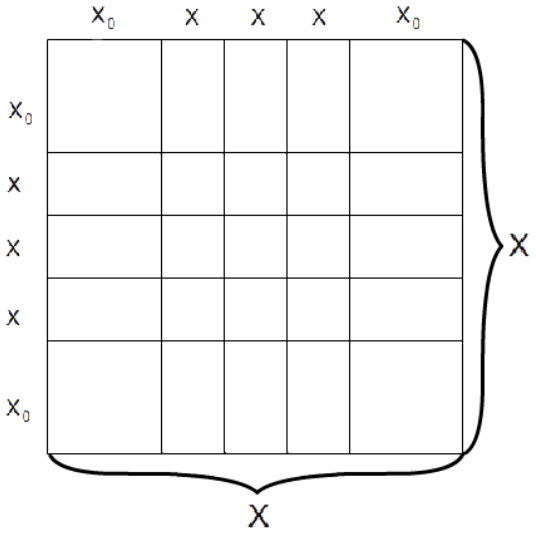


Рис. П1.1. Разбиение квадратного изображения на фрагменты

Ясно, что неравенство (П1.1) выполняется также для областей, расположенных на границах исходной области, т.к. они имеют меньшую длину сопряженных границ. Области, находящиеся в углах изображения размером $x_0 \times x_0$, будем называть угловыми, области $x_0 \times x$ ($x \times x_0$) – граничными, а области $x \times x$ – внутрен-

ними. Задача заключается в том, чтобы найти x_0 и x такие, чтобы время обработки всех областей с учетом затрат на пересылку было одинаковым (рис. П1.1).

Для простоты полагаем, что ширина полос данных на границах областей, которыми они должны обмениваться, равна одному отсчету, поэтому объем передаваемых данных пропорционален длине границ. Тогда суммарное время обработки с учетом пересылок:

а) для внутренней области:

$$T_{вн} = x^2 \cdot \tau_p + 4 \cdot x \cdot \tau_n, \qquad (П1.2)$$

б) для граничной:

$$T_{гр} = x \cdot x_0 \cdot \tau_p + 2 \cdot x_0 \cdot \tau_n + x \cdot \tau_n, \qquad (П1.3)$$

в) для угловой:

$$T_{угл} = x_0^2 \cdot \tau_p + 2 \cdot x_0 \cdot \tau_n. \qquad (П1.4)$$

Положим

$$x_0 = k \cdot x, \qquad (П1.5)$$

где $1 < k < 1{,}5$ – коэффициент увеличения угловой (и соответственно граничной) области, который необходимо выбрать из условия балансировки процессоров.

При балансировке процессоров, обрабатывающих внутренние и граничные области, вычислительные затраты на обработку угловых областей существенно возрастают. Поэтому потребуем, чтобы выполнялось равенство

$$T_{угл} = T_{вн}$$

или

$$k^2 \cdot x + 2 \cdot k \cdot \delta = x + 4 \cdot \delta. \qquad (П1.6)$$

Отбрасывая из решений (П1.6) отрицательные значения k, получаем

$$k = \frac{\delta}{x} + \sqrt{\frac{\delta}{x} + 1 + \frac{4 \cdot \delta}{x}}. \qquad (П1.7)$$

С учетом неравенства (П1.1) в соответствии с (П1.7) можно записать условие для допустимых значений k:

$$k \le -\frac{\Delta_{доп}}{4} + \sqrt{\left(\frac{\Delta_{доп}}{4}\right)^2 + 1 + 4 \cdot \Delta_{доп}} \,. \qquad \text{(П1.8)}$$

Остается подобрать удовлетворяющее условию (П1.8) наибольшее значение k, при котором целое число n (число полос, на которые разбивается область решений) удовлетворяет равенству

$$(n-2)x + 2kx = X \,. \qquad \text{(П1.9)}$$

Заметим, что обычно отношение δ/x невелико, при этом для значений k, удовлетворяющих (П1.8), время обработки граничных областей не превышает времени обработки угловых и внутренних областей. Соотношения (П1.8), (П1.9) могут использоваться для выбора начального разбиения исходной области на фрагменты.

В действительности эффективность загрузки процессоров будет зависеть от многих других факторов, которые не учитывались в нашей упрощенной модели (например, латентность при передаче данных, а также тот факт, что исходная область может быть не квадратной, а X не обязано делиться без остатка на величину x и др.). Для более полного учета влияния всех факторов, которые не принимались во внимание в указанной упрощенной постановке, может использоваться технология итерационного планирования распределения ресурсов, описанная в работе [9]. В данном случае реализация этой технологии не вызовет затруднений, поскольку задача выбора k однопараметрическая.

Приложение 2. Пример построения графа алгоритма с управляющими связями

Для построения взвешенного направленного графа с управляющими связями рассмотрим конкретную задачу вычисления массива данных, являющихся дискретными значениями переходного процесса в системе, описываемой дифференциальным уравнением второго порядка с правой частью в виде полинома второго порядка.

Решение дифференциального уравнения

$$\left(a_0 p^2 + a_1 p + a_2\right) \cdot x(t) = b_0 t^2 + b_1 t + b_2, \qquad (П2.1)$$

где $p = d/dt$ – алгебраизированный оператор дифференцирования, $x(t)$ – искомый процесс, а $a_0, a_1, a_2, b_0, b_1, b_2$ – заданные коэффициенты, представляют собой сумму его частного интеграла $x_ч(t)$ и общего интеграла $x_о(t)$ соответствующего однородного уравнения. Частное решение уравнения (П2.1) в предположении, что система имеет запас устойчивости (т.е. вещественная часть корней характеристического уравнения не равна нулю), имеет вид

$$x_ч(t) = A_0 t^2 + A_1 t + A_2, \qquad (П2.2)$$

где
$$\left.\begin{aligned} A_0 &= b_0 / a_2, \\ A_1 &= \left(b_1 a_2 - 2 b_0 a_1\right) / a_2^2, \\ A_2 &= \left(b_1 a_2^2 - 2 a_0 b_0 a_2 - a_1 b_1 a_2 + 2 b_0 a_1^2\right) / a_2^3. \end{aligned}\right\}$$

Возможные решения дифференциального уравнения

$$x_0(t) = C_1 e^{-\alpha_1 t} + C_2 e^{-\alpha_2 t}, \text{ где } x_0(t) = C_1 = \frac{\alpha_2 x_0 + x_0}{} e^{-\alpha_1 t} + C_2 e^{-\alpha_2 t}. \qquad (П2.3)$$

Выражения для вычисления значений составляющей процесса $x_о(t)$ при различных типах корней характеристического уравнения сведены в табл. П2.1. Здесь α, α_1, α_2, β – абсолютные значения соответствующих величин.

Алгоритм решения задачи можно представить в виде, показанном на рис. П2.1. Здесь вид некоторых формул изменен по сравнению с таблицей из методических соображений, что, впрочем, может оказаться целесообразным и по существу.

На схеме отражены два типа операторов: логические (ромбом) и арифметические (прямоугольником). Логические определяют связи по управлению.

Они задают состав и порядок выполнения операторов. Прочие операторы определяют только порядок выполнения операторов, определяемый информационными связями между ними.

Таблица П2.1

Вещественные некратные корни α_1, α_2	Вещественный кратный корень α	Пара комплексных корней $\alpha \pm j\beta, \alpha, \beta$
$x_o(t) = C_1 e^{-\alpha_1 t} + C_2 e^{-\alpha_2 t}$, где $C_1 = \frac{\alpha_2 x_0 + x_0'}{\alpha_2 - \alpha_1}$, $C_2 = \frac{\alpha_1 x_0 + x_0'}{\alpha_1 - \alpha_2}$.	$x_o(t) = (C_1 + C_2 t) e^{-\alpha t}$, где $C_1 = x_0$, $C_2 = x_0' + \alpha x_0$.	$x_o(t) = (C_1 \cos \beta t + C_2 \sin \beta t) e^{-\alpha t}$, где $C_1 = x_0$, $C_2 = (x_0' + \alpha x_0)/\beta$.

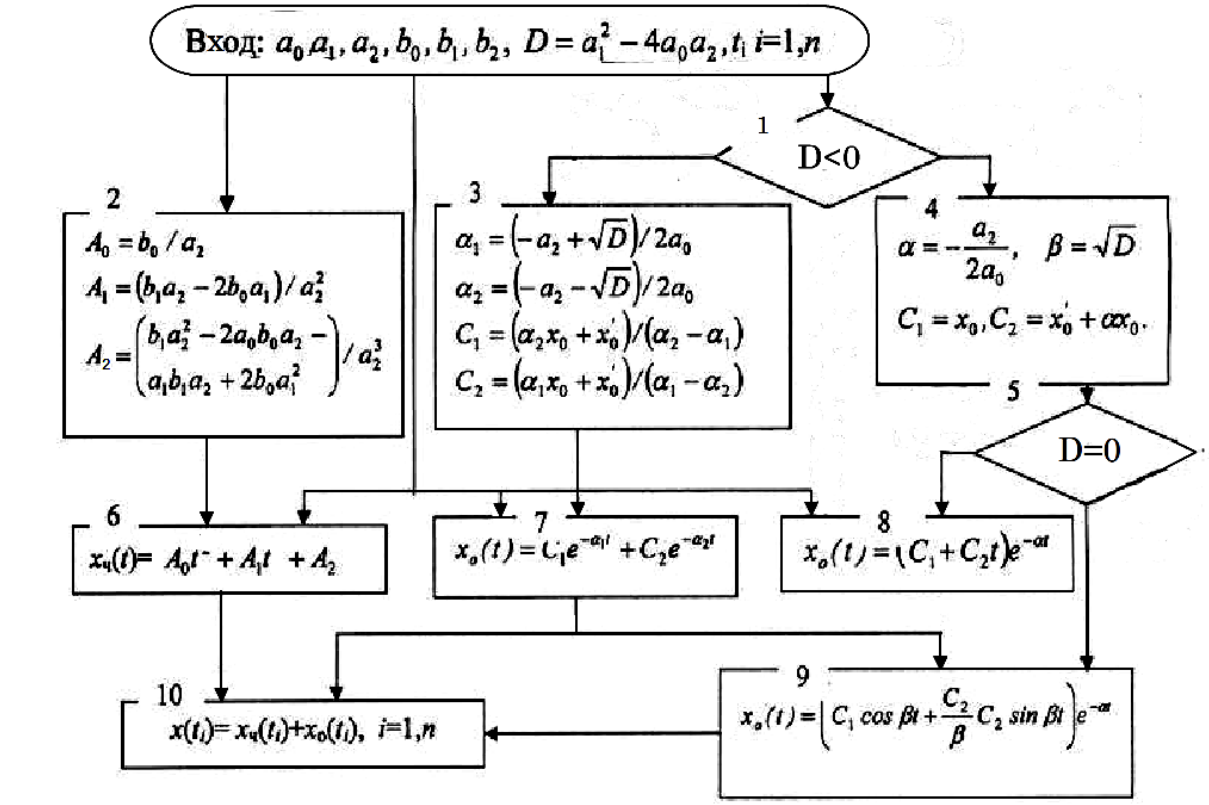


Рис. П2.1. Блок-схема алгоритма

Возможность распараллеливания алгоритмов зависит от степени детализации операторов. Например, на приведенной схеме любой из блоков, включающий совокупность арифметических выражений, может быть представлен в виде параллельных арифметических операторов. Далее каждый блок, помеченный цифрой, мы будем называть оператором, независимо от того, какую совокупность соотношений или задач он представляет.

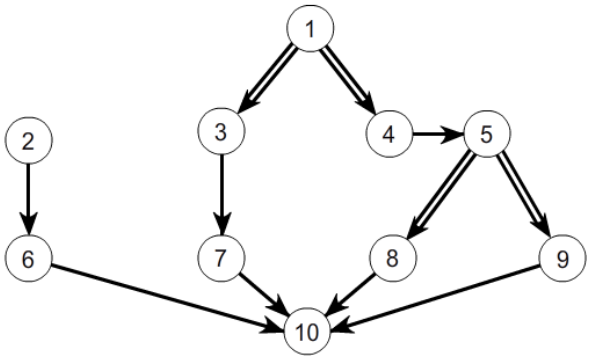


Рис. П2.2. Граф-схема алгоритма

Соответствующая блок-схеме алгоритма (рис. 5.1) граф-схема приведена на рис. П2.2. Здесь множество $\mathbf{X} = \{i\} = \{\overline{i = 1,m}\}$ вершин графа соответствует множеству операторов алгоритма. Множество дуг состоит из двух типов, определяющих связи по управлению (двойные стрелки) и по информации (обычные стрелки).

Переход от обычной блок-схемы алгоритма к модели в виде граф-схемы дает более ясное представление о структуре алгоритма, его свойствах и возможности направленных преобразований. Заметим, что каждая связь по управлению одновременно является связью по информации, т.к. она определяет строгий порядок следования операторов, задаваемый логической переменной. В простых случаях выявить операторы, которые могут выполняться независимо и параллельно, можно непосредственно по граф-схеме алгоритма. Если граф-схема имеет большое число ветвей, такой анализ становится затруднительным. В этом случае анализ граф-схем можно проводить с использованием формальных правил преобразований соответствующих матриц.

Приложение 3. Анализ параллелизма алгоритмов с использованием матричного представления графа

Методику анализа параллелизма алгоритмов с использованием матричного представления графа рассмотрим на примере построенного выше в приложении 2 графа алгоритма с управляющими связями (рис. П2.2). Этот граф обладает вычислительной простотой и вместе с тем содержит разветвления. Это позволит нам на простом примере наглядно показать все основные правила преобразования графов.

Для указанного графа введем в рассмотрение матрицу следования S по следующим правилам. Вершине (оператору) i графа поставим в соответствие i-ю строку и i-й столбец матрицы. Элемент (i,j) этой матрицы будем отмечать знаком •, если между операторами с этими номерами существует связь вида $j \Rightarrow i$ (по управлению или информации). Для различения типа операторов там, где это необходимо, связи по управлению будем отмечать знаком **C**, а связи по информации – **I**.

Можно показать [1], что по графу без контуров может быть построена треугольная матрица следования с нулевыми элементами на главной диагонали, что упрощает некоторые задачи их анализа. На рис. П3.1 приведена матрица следования, соответствующая граф-схеме, показанной на рис. П2.2.

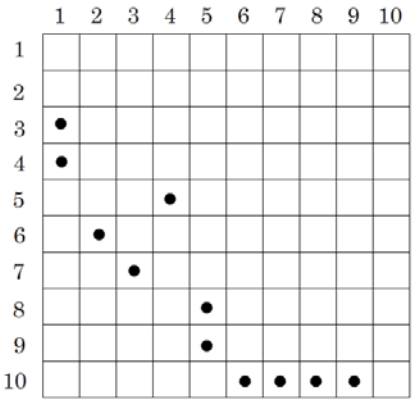


Рис. П3.1. Матрица следования S

Для дальнейшего исследования матрицы S необходимо отразить в ней (неявные) связи между операторами, которые реально существуют и определяются задающими связями, но непосредственно между ними в графе отсутствуют. Их введение необходимо для выявления ветвей связанных операторов, которые не могут выполняться параллельно. Кроме того, введение этих связей позволяет выявить контуры в графе. Рассмотрим способы установления этих связей.

Если существуют задающие связи $\beta \to \gamma$ и $\gamma \to \delta$, но нет задающей связи $\beta \to \delta$, то дополнение вычислительной схемы такими связями не меняет результата решения задачи, а лишь подтверждает недопустимость определенного порядка следования операций. Связи, которые введены направленно внутри всех пар операторов, принадлежащих одному пути в графе *G*, и не связаны задающими связями, образуют множество транзитивных связей [1]. Транзитивные связи могут быть введены путем формальных преобразований матрицы *S* по следующему алгоритму.

Строки матрицы просматриваются последовательно сверху вниз. В каждой (*i*-й) строке просмотр элементов производится в порядке увеличения номеров столбцов. Если в очередном (*j*-м) столбце имеется знак •, указывающий на существование связи по информации или управлению, одноименные элементы строк с номерами *i* и *j* складываются по правилу дизъюнкции (или): •∨•=•, •∨0=•, 0∨•=•, 0∨0=0, т.е. в просматриваемую строку добавляются все знаки •, которые имеются в *j*-й строке. На рис. П3.2 приведена треугольная матрица следования, полученная из матрицы, показанной на рис. П3.1, путем ее дополнения транзитивными связями.

	1	2	3	4	5	6	7	8	9	10
1										
2										
3	•									
4	•									
5	•			•						
6		•								
7	•		•							
8	•			•	•					
9	•			•	•					
10	•	•	•	•	•	•	•	•	•	

Рис. П3.2. Матрица следования *S*, дополненная транзитивными связями

Заметим, что, если исходная матрица не треугольная, это может свидетельствовать о наличии контуров в графе. Для их выявления просмотр строк матрицы *S* производится неоднократно до установления факта ее неизменности. В результате указанных преобразований вводятся транзитивные связи, с помощью которых устанавливается факт наличия контура в информационно-логическом графе. О наличии контуров свидетельствуют ненулевые диагональные элементы.

Для того чтобы в явном виде показать, какие операторы не могут выполняться параллельно, необходимо использовать только информационные связи. Дело в том, что эти связи прямо указывают на то, какие операторы не могут выполняться одновременно. Например, в схеме на рис. П2.2 операторы 3, 4 не

могут выполняться не только одновременно, но даже при одной реализации алгоритма. Такие операторы называются логически несовместимыми. Для выявления таких операторов используются матрицы L – логической несовместимости. Рассмотрим методику их формирования.

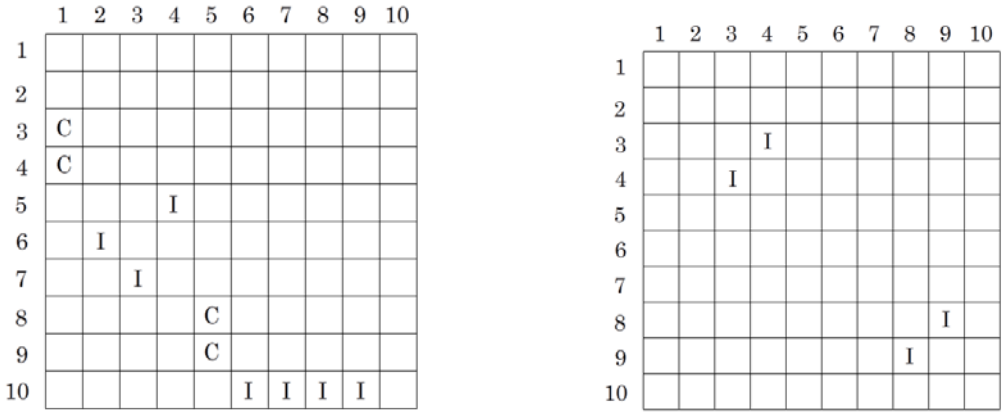
Вначале по исходной треугольной матрице S, в которой связи по управлению отмечены знаком **C**, а связи по информации – **I**, формируются задающие связи логической несовместимости операторов по следующим правилам. Последовательно просматривают столбцы $j = 1,m$ матрицы S. Если очередной элемент (i_μ,j) = **C** и ранее в этом столбце также были зафиксированы элементы (i_k,j) = **C**, $k = 1,\mu - 1$, то все элементы в i_μ-й строке с номером, совпадающим с номером строки, в которой ранее встречался знак **C**, заполняются единицами до i_μ-1-го столбца включительно. Для каждой заполненной описанным способом единицы затем в матрицу вносится симметричная относительно главной диагонали единица. Матрица следования граф-схемы алгоритма (рис. П2.2) с указанием отдельно связей по управлению и информации и соответствующая ей матрица логической несовместимости операторов L приведены на рис. П3.3.

Далее на основе заданных связей логической несовместимости определяется несовместимость для всех операторов схемы. Для этого вводятся транзитивные связи логической несовместимости по следующим правилам.

Последовательно просматривают строки треугольной матрицы следования S, дополненной транзитивными связями (нулевые строки пропускают). В очередной ненулевой i-й строке матрицы S анализируют множество ненулевых элементов. Из этого множества исключают номера операторов, образующих входы матрицы S, т.е. обнуляют элементы в столбцах, номера которых совпадают с номерами нулевых строк.

С использованием оставшегося множества номеров операторов (ненулевых элементов матрицы S) последовательно, начиная с первой строки, формируют строки i^*. Для этого объединяют по операции конъюнкции (И) строки матрицы L с номерами, соответствующими номерам столбцов матрицы S, в которых остались ненулевые элементы. Если ненулевой элемент в строке один, то строку

i^* полагают равной строке матрицы L с номером, равным номеру столбца ненулевого элемента в анализируемой строке.



а) б)

Рис. П3.3. Исходная матрица следования (а) и соответствующая ей матрица логической несовместимости операторов (б)

Далее обновляют матрицу L . Новую i-ю строку в матрице L формируют на основе ее старого значения и вновь сформированной строки i^* с помощью операции дизъюнкции $i = i \vee i^*$.

После обновления очередной строки матрицы L i-й столбец матрицы L полагают равным вновь сформированной i-й строке. Далее с использованием обновленной матрицы формируют очередную строку i^* по правилу, описанному выше. Процесс формирования продолжается до исчерпания строк.

На рис. П3.4 для используемого здесь примера показано построение матрицы L логической несовместимости операторов с транзитивными связями в соответствии с описанными правилами. Матрица логической несовместимости операторов, дополненная транзитивными связями логической несовместимости, позволяет судить о том, могут или не могут конкретно взятые операторы выполняться при одной реализации алгоритма.

Заключительный этап анализа параллелизма алгоритма заключается в объединении информации о логической несовместимости операторов и их информационно-логической связи по следующим правилам. Вначале по матрице S, дополненной транзитивными связями, строят матрицу следования S' путем симметричного отображения элементов матрицы S относительно главной диа-

гонали (т.е. путем сложения исходной и транспонированной матриц). Далее на сформированную матрицу S' накладывают матрицу L, дополненную транзитивными связями. Каждый элемент новой матрицы M формируется из одноименных элементов матриц S' и L по правилу дизъюнкции. Нулевые элементы полученной матрицы M указывают множество тех операторов, каждый из которых при выполнении некоторых условий может быть выполнен одновременно с данным, т.е. он информационно или по управлению не зависит от этого оператора и не является с ним логически несовместимым. На рис. П3.5 приведены построенные по указанному правилу матрицы S' и M.

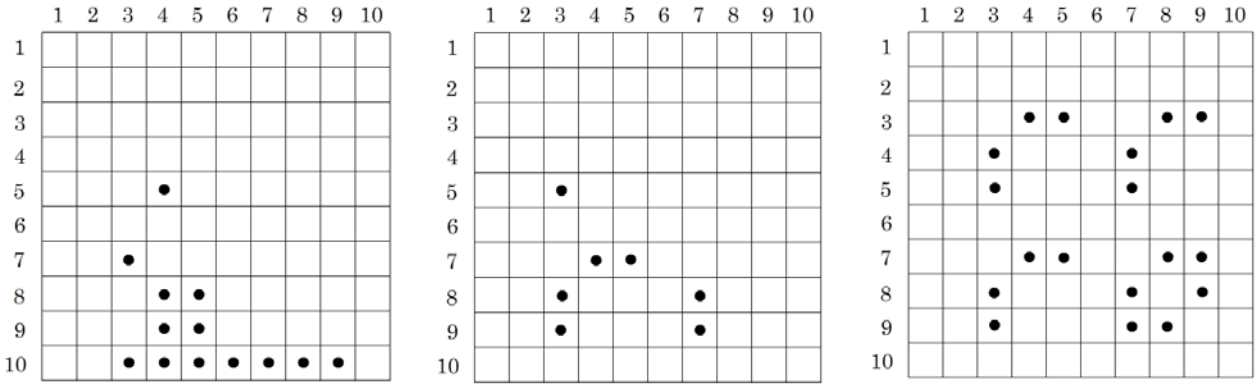
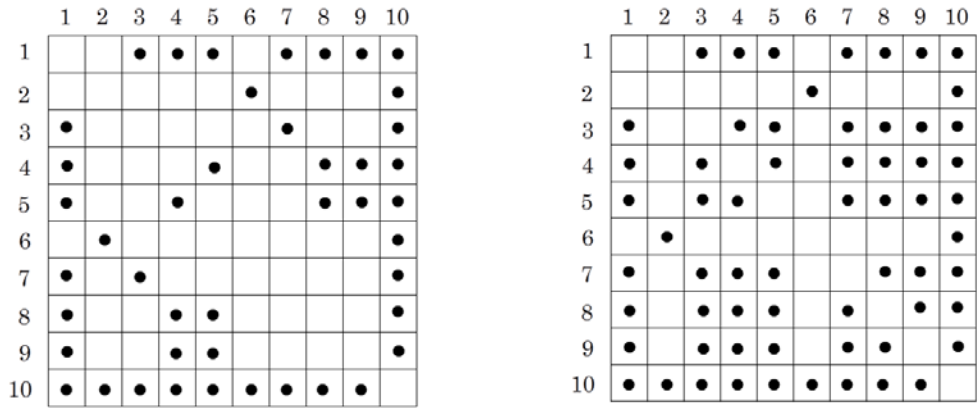


Рис. П3.4. Дополнение матрицы логической несовместимости транзитивными связями:
а – матрица S с исключенными входами; б – матрица, составленная
из строк i^*; в – обновленная матрица логической несовместимости

Симметричную матрицу M называют матрицей независимости. Она отражает информационно-логические связи между операторами без учета их ориентации, а также связи логической несовместимости операторов с учетом всех транзитивных связей. Например, из матрицы M следует, что оператор 2 может выполняться независимо, а следовательно, если необходимо, то и параллельно с операторами 5 и 8, однако нельзя говорить о *взаимной независимости операторов* (ВНО), так как существует связь $5 \Rightarrow 8$. Целью анализа параллелизма алгоритма является выявление множеств ВНО, а также нахождение максимально полного множества ВНО, содержащего максимальное число операторов.



а) б)

Рис. П3.5. Матрица следования S' (а) и соответствующая матрица независимости $M = S' \vee L$ (б)

Для его нахождения используется следующее свойство. Если два оператора μ и ν взаимно независимы, то при сложении по правилам дизъюнкции μ-й и ν-й строк матрицы независимости M образуется строка, в которой элементы в μ-м и ν-м столбцах обязательно нулевые. Заметим, что если не рассматриваются связи по управлению, то матрица независимости M совпадает с матрицей следования S, дополненной транзитивными связями. Например, при сложении десятой строки матрицы M, представленной на рис. П3.8*б* (по правилу дизъюнкции), с остальными строками получаются строки, содержащие все единичные элементы. Следовательно, оператор 10 образует полное множество ВНО. При сложении строк, соответствующих операторам 1-м и 2-м, получается строка с нулевыми элементами в 1-м и 2-м столбцах. Следовательно, {1,2} – полное множество ВНО с числом элементом, превышающим ранее найденное. Множества с двумя элементами образуют также операторы {1,6}, {2,3}, {2,4}, {3,6}, {4,6}, {5,6}, {6,7}, {6,8}, {6,9}. Установленные полные множества ВНО содержат не более двух элементов. При сложении трех и более строк матрицы M в любых сочетаниях не удается получить строку, в которой элементы с номерами, совпадающими с номерами складываемых строк, оказались нулевыми. Следовательно, множество ВНО с двумя элементами является максимально полным множеством ВНО.

Приложение 4. Анализ производительности вычислительной системы по графу алгоритма

Предположим, дуга графа системы идет из *i*-го устройства в *j*-е. Поскольку результат *i*-го устройства является аргументом *j*-го, количество операций, выполняемых *j*-м устройством, не может более чем на 1 отличаться от количества операций, реализованных *i*-м устройством:

$$N_i - 1 \le N_j \le N_i + 1. \qquad (П4.1)$$

Допустим, связный граф содержит q дуг. Если *k*-е устройство за время T выполнило N_k операций, а *l*-е – N_l операций, то из (П4.1) вытекает, что

$$N_l - q \le N_k \le N_l + q$$

для любых $k, l, 1 \le k, l \le s$.

Перенумеруем устройства так, что $\pi_1 \le \pi_2 \le \ldots \le \pi_s$. Тогда в соответствии с последним неравенством можно записать

$$(N_1 - q)s + q \le \sum_{i=1}^{s} N_i \le (N_1 + q)s - q. \qquad (П4.2)$$

Разделив все части неравенств (8.7) на T с учетом того, что

$$\frac{1}{T}\sum_{i=1}^{s} N_i = r, \quad \frac{N_1}{T} = \pi_1,$$

указанные неравенства можно переписать в виде

$$\pi_1 s - \frac{q(s-1)}{T} \le r \le \pi_1 s + \frac{q(s-1)}{T}. \qquad (П4.3)$$

Слагаемые $q(s-1)/T$ в неравенствах (П4.3) при увеличении T стремятся к нулю. Это означает, что для системы из s устройств с пиковыми производительностями $\pi_1, \ldots, \pi_s$, описываемой связным графом, максимальная производительность $r_{\max}$ определяется как

$$r_{\max} = s \min_{1 \le i \le s} \pi_i. \qquad (П4.4)$$

Приложение 5. Доказательство 2-го закона Амдала

Если пиковые производительности всех устройств одинаковы и равны π, в соответствии с (8.1) – (8.3) ускорение определяется как

$$R = \sum_{i=1}^{s} p_i \,. \qquad (П5.1)$$

Загруженность устройства, на котором выполняется последовательная часть программы, равна единице. Загруженности остальных устройств

$$p_i = \frac{(1-\beta)N/s}{\beta N + (\beta - 1)N/s}, \quad i = \overline{2,s}\,.$$

Следовательно, в соответствии с (П5.1)

$$R = 1 + \sum_{i=2}^{s} \frac{(1-\beta)N/s}{\beta N + (1-\beta)N/s} = \frac{s}{\beta s + (1-\beta)}\,.$$

Приложение 6. Доказательство закона Густавсона–Барсиса

Для доказательства воспользуемся обозначением доли последовательных расчетов в виде (8.12):

$$g = \frac{\tau_n}{\tau_n + \tau_{N-n}/s}, \qquad (П6.1)$$

где τ_n и τ_{N-n} – время, необходимое для выполнения последовательной и параллельной частей соответственно.

С учетом введенных обозначений время решения задачи на одном и s процессорах соответственно

$$T_1 = \tau_n + \tau_{N-n}, \quad T_{s1} = \tau_n + \tau_{N-n}/s. \qquad (П6.2)$$

С другой стороны, из соотношения (П6.1) для величины g можно записать:

$$\tau_n = g(\tau_n + \tau_{N-n}/s), \qquad \tau_{N-n} = (1-g)s(\tau_n + \tau_{N-n}/s). \qquad (П6.3)$$

С учетом (П6.1), (П6.2) и (П6.3) получаем оценку для ускорения

$$R = \frac{T_1}{T_s} = \frac{\tau_n + \tau_{N-n}}{\tau_n + \tau_{N-n}/s} = \frac{(g + (1-g)s)(\tau_n + \tau_{N-n}/s)}{\tau_n + \tau_{N-n}/s} = g + (1-g)s. \qquad (П6.4)$$

Оценку (П6.4) называют *законом Густавсона–Барсиса*. Нетрудно заметить, что эту оценку можно также переписать в виде:

$$R = \frac{T_1}{T_s} = s + (1-s)g. \qquad (П6.5)$$

Приложение 7. Построение соотношений для прогноза ускорения и эффективности реализации параллельного алгоритма умножения матрицы на вектор

Рассматривается задача умножения m×n-матрицы на n×1-вектор. Эта задача сводится к вычислению m скалярных произведений векторов длины n. Каждое такое произведение требует n операций умножения и $n-1$ операций сложения. Таким образом, вычислительная сложность последовательного алгоритма составит $T_1 = m(2n-1)$, а в случае квадратной n×n-матрицы –

$$T_1 = n(2n-1). \qquad (П7.1)$$

Если умножение n×n-матрицы выполняется параллельно (рис. 7.1), за каждым процессором закрепляется не более

$$m_s =]n/s[\qquad (П7.2)$$

строк, где $]*[$ – здесь и далее означает операцию округления до целого в большую сторону. Количество процессоров, за которыми будет закреплено меньше, чем m_s строк, определяется конкретными значениями n и s.

Для построения оценок ускорения и эффективности с учетом затрат на вычисления и коммуникации предполагаем, что все $(2n-1)$ операций умножения и сложения имеют одинаковую длительность τ, а вычислительная система однородна, т.е. все процессоры обладают одинаковой производительностью. Тогда временные затраты параллельного алгоритма, связанные непосредственно с вычислениями, с учетом (П7.2) составят

$$T_s = \left]\frac{n}{s}\right[(2n-1)\cdot\tau. \qquad (П7.3)$$

Если сеть передачи данных имеет структуру гиперкуба или полного графа, операция сбора данных может быть выполнена за $\lceil log_2 s \rceil$ итераций. На первой итерации взаимодействующие пары процессоров обмениваются сообщениями объемом $w(n/s)$, где w – размер одного элемента вектора в байтах. На второй итерации этот объем увеличивается вдвое и оказывается равным $2w\lceil n/s \rceil$ и т.д.

Общая длительность сбора данных

$$T_s = \sum_{i=1}^{]\log_2 s[} \left(\alpha + 2^{i-1} w\right] n/s[/\beta\right) = \alpha\,]\log_2 s[+ w]n/s[\left(\sum_{i=1}^{]\log_2 s[} 2^{i-1}\right)/\beta\,, \qquad (П7.4)$$

где α – латентность сети передачи данных, β – пропускная способность сети.

Можно показать, что

$$\sum_{i=1}^{]\log_2 s[} 2^{i-1} = \left(2^{]\log_2 s[} - 1\right) = s - 1. \qquad (П7.5)$$

С учетом (П7.3), (П7.4) и (П7.5) общее время выполнения параллельного алгоритма

$$T_s = \left]\frac{n}{s}\right[\left(2n-1\right)\cdot\tau + \alpha\,]\log_2 s[+ w]n/s[\left(s-1\right)/\beta\,. \qquad (П7.6)$$

Если n/s и $log_2\, s$ – целые числа, в соответствии с (П7.1), (П7.6) оценки для ускорения и эффективности принимают вид

$$R = \frac{T_1}{T_s} = \frac{n\left(2n-1\right)\cdot\tau}{\left(n/s\right)\left(2n-1\right)\cdot\tau + \alpha\left(log_2\, s\right) + w\left(n/s\right)\left(s-1\right)/\beta}, \qquad (П7.7)$$

$$E_s = \frac{R}{s} = \frac{n\cdot\left(2n-1\right)\cdot\tau}{n\cdot\left(2n-1\right)\cdot\tau + s\left\{\alpha\left(\log_2 s\right) + w\left(n/s\right)\left(s-1\right)/\beta\right\}}. \qquad (П7.8)$$

Из соотношений (П7.8) видно, что для сохранения требуемого уровня эффективности при увеличении числа используемых процессоров соответственно должна возрастать вычислительная сложность задачи (размерности матрицы и вектора). Заметим также, что, если в (П7.7), (П7.8) пренебречь затратами на передачу данных, будем иметь

$$R = T_1/T_s = s\,, \qquad E_s = R/s = 1.$$

Высокая (предельно возможная) степень параллелизма задачи в данном случае связана с тем, что отсутствуют операции, которые не поддаются распараллеливанию.

Приложение 8. Текст программы `Multiply.c`

```
#include "mpi.h"
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char *argv[])
{
    int my_rank;
    int rank_size;

    MPI_Init(&argc, &argv); // Важно!
    MPI_Status status;

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    int nsize, msize;

    ifstream f_in_a (argv[1]);
    ifstream f_in_x (argv[2]);
    ofstream f_out_y (argv[3]);
    f_in_a >> nsize;
    f_in_a >> msize;


    double a[nsize][msize];
    double x[msize];
    double y[nsize];

    for (int i = 0; i < nsize; i++){
        for (int j = 0; j < msize; j++){
            f_in_a >> a[i][j];
        }
    }

    for (int j = 0; j < msize; j++){
        f_in_x >> x[j];
    }
```

```
    int start = (my_rank*nsize)/rank_size;
    int final = ((my_rank+1)*nsize)/rank_size;

    for (int i = start; i < final; i++){
        y[i] = 0.0;
        for (int j = 0; j < msize; j++){
            y[i] += a[i][j]*x[j];
        }
    }

    int tmp[2];
    if (my_rank%2 == 1){
      tmp[0] = start;
      tmp[1] = final;
      MPI_Send(&tmp, 2, MPI_INT, my_rank - 1, 0, MPI_COMM_WORLD );
      MPI_Send(&y[tmp[0]], tmp[1]- tmp[0], MPI_DOUBLE, my_rank - 1,
0, MPI_COMM_WORLD );
    }
    if (my_rank%2 == 0) {
      MPI_Recv(&tmp, 2, MPI_INT, my_rank + 1, 0, MPI_COMM_WORLD,
&status );
      MPI_Recv(&y[tmp[0]], tmp[1] - tmp[0], MPI_DOUBLE, my_rank +
1, 0, MPI_COMM_WORLD, &status );
    }

    if (my_rank%4 == 2){
        tmp[0] = start;
      MPI_Send(&tmp, 2, MPI_INT, my_rank - 2, 0, MPI_COMM_WORLD );
      MPI_Send(&y[start], tmp[1] - tmp[0], MPI_DOUBLE, my_rank - 2,
0, MPI_COMM_WORLD );
    }
    if (my_rank%4 == 0){
        MPI_Recv(&tmp, 2, MPI_INT, my_rank + 2, 0, MPI_COMM_WORLD,
&status );
      MPI_Recv(&y[tmp[0]], tmp[1] - tmp[0], MPI_DOUBLE, my_rank +
2, 0, MPI_COMM_WORLD, &status );
    }

    if (my_rank%8 == 4){
        tmp[0] = start;
      MPI_Send(&tmp, 2, MPI_INT, my_rank - 4, 0, MPI_COMM_WORLD );
```

```
      MPI_Send(&y[start], tmp[1] - tmp[0], MPI_DOUBLE, my_rank - 4,
0, MPI_COMM_WORLD );
    }
    if (my_rank%8 == 0){
        MPI_Recv(&tmp, 2, MPI_INT, my_rank + 4, 0, MPI_COMM_WORLD,
&status );
      MPI_Recv(&y[tmp[0]], tmp[1] - tmp[0], MPI_DOUBLE, my_rank +
4, 0, MPI_COMM_WORLD, &status );
    }

    if (my_rank == 0){
     for (int i = 0; i < nsize; i++){
            f_out_y << y[i] << endl;
        }
    }

    MPI_Finalize(); // Важно!

    return 0;
}
```

Список использованных источников

1. Барский, А.Б. Параллельные процессы в вычислительных системах. Планирование и организация / А.Б. Барский. – М.: Радио и связь, 1990. – 256 с.
2. Воеводин, В.В. Параллельные вычисления / В.В. Воеводин, Вл.В. Воеводин. – СПб.: БХВ-Петербург, 2002.
3. Гергель, В.П. Лекции по параллельным вычислениям: учеб. пособие / В.П. Гергель, В.А. Фурсов. – Самара: Изд-во СГАУ, 2009. – 164 с.
4. Гладких, Б.А. Информатика от абака до Интернета. Введение в специальность: учеб. пособие / Б.А. Гладких. – Томск: Изд-во НТЛ, 2005. – 484 с.
5. Головашкин, Д.Л. Методы параллельных вычислений: учеб. пособие / Д.Л. Головашкин. – Самара: Изд-во СГАУ, 2002. Ч. I. – 92 с.
6. Головашкин, Д.Л. Методы параллельных вычислений: учеб. пособие / Д.Л. Головашкин, С.П. Головашкина. – Самара: Изд-во СГАУ, 2003. Ч. II. – 103 с.
7. Введение в программирование для параллельных ЭВМ и кластеров: учеб. пособие / В.В. Кравчук, С.Б. Попов, А.Ю. Привалов [и др.]. – Самара: Самар. науч. центр РАН, Самар. гос. аэрокосм. ун-т, 2000. – 87 с.
8. Тоффоли, Т. Машины клеточных автоматов / Т. Тоффоли, Н. Марголус. – М.: Мир, 1991.
9. Фурсов, В.А. Технология итерационного планирования распределения ресурсов гетерогенного кластера / В.А. Фурсов, В.А. Шустов, С.А. Скуратов // Тр. Всерос. науч. конф. «Высокопроизводительные вычисления и их приложения»; Тр. Всерос. науч. конф. г. Черноголовка, 30 октября – 2 ноября 2000 г. – С. 43-46.
10. Bruce, P. Lester. The art of parallel programming / P. Bruce. – Prentice Hall, Englewood Cliffs, New Jersey, 1993. – 376 p.
11. Computational Science: Ensuring America`s Competitiveness. President`s Information Tehnology Advisory Committee. May 27, 2005.
12. Букатов, А.А. Программирование многопроцессорных вычислительных систем / А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. – Изд-во ООО «ЦВВР», 2003.

Учебное издание

Фурсов Владимир Алексеевич

ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА КЛАСТЕРЕ

Учебное пособие

Редактор Ю.Н. Литвинова
Доверстка Е.С. Кочеулова

Подписано в печать 16.08.15. Формат 60×84 1/16.
Бумага офсетная. Печать офсетная. Печ. л. 5,0.
Тираж 200 экз. Заказ . Арт. – 34/2015.

федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский государственный аэрокосмический
университет имени академика С. П. Королева
(национальный исследовательский университет)» (СГАУ)
443086, Самара, Московское шоссе, 34.

Изд-во СГАУ. 443086, Самара, Московское шоссе, 34.

ДЛЯ ЗАМЕТОК

ДЛЯ ЗАМЕТОК