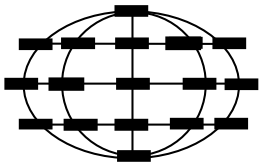
Новосибирский государственный
технический университет

В.Э.Малышкин, В.Д.Корнеев

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ МУЛЬТИКОМПЬЮТЕРОВ



Новосибирск – 2006

ББК 32.973-018.1
УДК 681.3.06

Рецензенты
Кафедра Вычислительных систем
Новосибирского государственного университета

Б.Г.Глинский, д.т.н., профессор

Кафедра Параллельных вычислительных технологий
Новосибирского государственного университета

В.А.Вшивков, д.ф.-м.н., профессор

В книге рассмотрены основные понятия параллельного программирования мультикомпьютеров, приведены краткие обзоры основного на текущий момент инструмента параллельного программирования мультикомпьютеров – библиотеку MPI, и архитектур современных микропроцессоров и вычислительных систем. Книга содержит материалы курсов по параллельному программированию, которые в течении 10 лет читаются в Новосибирском государственном техническом университете для студентов факультета Прикладной математики и информатики и в Новосибирском государственном университете для студентов факультета Информационных технологий.

Для студентов, магистрантов и аспирантов вузов, для научных сотрудников и практикующих программистов, для всех желающих изучить параллельное программирование с самого начала.

Подготовка книги поддержана из средств программы Рособразования "Развитие научного потенциала ВШ", проект РНП.2.2.1.1.3653.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В книге приводятся и обсуждаются базисные понятия из области параллельных вычислений и параллельного программирования мультикомпьютеров.

Суперкомпьютеры позволили начать решать новые задачи большого размера, приступить к реализации реалистических математических моделей сложных физических явлений и технических устройств, которые раньше нельзя было решать на маломощных последовательных компьютерах. В этой области суперкомпьютеры являются незаменимым и универсальным технологическим инструментом как в изучении природы, так и в практической деятельности. Понятна привлекательность изучения ядерного взрыва на модели, а не в натуре.

Использование суперкомпьютеров наталкивается на большие трудности. Очень сложно создавать и отлаживать параллельные программы, гораздо сложнее последовательных. В самых простых с виду параллельных программах обнаруживаются иной раз фатальные ошибки. Известны программы, содержащие буквально десяток операторов, которые долгое время считались правильными. И тем не менее со временем в них обнаруживали тяжелые ошибки! А ведь среди приложений суперкомпьютеров есть много и таких, которые очень чувствительны к ошибкам. Даже незначительные ошибки в программе могут приводить к тяжелым последствиям. Отсюда

значительно бóльший, чем в последовательном программировании, интерес к автоматизации конструирования параллельных программ, к строгим математическим методам конструирования и анализа правильности программ.

Профессия программиста требует основательного знания математики. Незнание математики существенно ограничивает возможности понимания сложного поведения системы параллельно протекающих и взаимодействующих вычислительных процессов. Инженеры, не имеющие подготовки в математике, быстро оказываются в состоянии не полного понимания устройства параллельного программного обеспечения.

Теория параллельных вычислений характеризуется большим разнообразием идей, подходов к решению задач, к разработке параллельных программ. Часто задачи требуют учета их особенностей для разработки эффективных программ решения. Это в свою очередь приводит к разработке разнообразных и многочисленных специализированных средств параллельного программирования. Все средства невозможно (да и не нужно) изучить. По этой причине более продуктивно начать с изучения общих моделей и методов параллельного программирования, нежели с конкретных систем программирования

Чтобы избежать рассмотрения множества не существенных на первых порах деталей и освоить идеи параллельного программирования в достаточно общем виде, изучение предмета начнется на довольно высоком уровне абстракции. Это, конечно, потребует немалых усилий при изучении, но окупится со временем более глубоким пониманием природы параллельных вычислений. А потому здесь не будет рассматриваться все разнообразие идей параллельного программирования, а выбран путь, который быстро приведет к пониманию сути параллельного программирования. А это и есть основная цель книги – правильно сформировать исходную точку зрения на предмет.

Для лучшего понимания и усвоения теоретического материала необходимо, конечно, и знакомство с нынешними вычислительными реалиями. Потому в книге кратко рассматриваются и основные на сегодняшний день средства задания множества параллельно исполняющихся и взаимодействующих процессов (системы MPI и OpenMP), а также приводится обзор архитектур современных микропроцессоров и вычислительных систем.

Книга трактует вопросы именно параллельного программирования (в узком смысле). Имеется ввиду, что процесс решения задачи разбивается (довольно условно) на несколько крупных этапов: *«физическая» формулировка задачи* $\rightarrow$

математическая модель → дискретная (численная) модель → алгоритмы решения задачи → программирование как реализация алгоритма решения задачи. Именно это программирование и рассматривается. Самые важные проектные решения, в наибольшей степени влияющие на качество решения всей прикладной задачи, принимаются на этапе «физической» формулировки задачи, где она описывается в терминах предметной области. Следующий по важности этап – математическая модель и алгоритмизация решения задачи и лишь последний этап – программирование. Ошибки, допущенные на более раннем этапе уже непоправимы на последующих. Например, выбор плохого алгоритма решения задачи невозможно исправить сколь угодно хорошим программированием и, в частности, последовательный по природе алгоритм параллельно не исполнить. Поэтому максимальные усилия в решении задачи должны прикладываться на ранних этапах. Но надо также иметь ввиду, что неудачное программирование параллельного исполнения алгоритма может многократно (в десятки, сотни и тысячи раз) ухудшить качество решения задачи по сравнению со средне хорошей программой. И в параллельном программировании такое ухудшение очень легко допустить! Потому так важно знание методов параллельного программирования или хотя бы понимание проблемы.

Книга построена на материалах курсов по параллельному программированию, которые в разных вариантах читаются с 1996 года в Новосибирском государственном техническом университете для студентов факультета Прикладной математики и информатики и в Новосибирском государственном университете для студентов факультета Информационных технологий. Частями материалы книги были опубликованы в нескольких учебных пособиях. Список рекомендованной литературы приведен в конце книги. Конечно, исчерпывающе полной информации о программировании мультикомпьютеров книга не содержит, однако после её прочтения усердный читатель будет в состоянии самостоятельно разобраться с конкретными языками, системами и технологиями параллельного программирования и правильно их применить на практике.

Демонстрируя плюсы и минусы параллелизма, авторы кардинально распараллелили свою работу над книгой: главы 1-8 написаны В.Э.Малышкиным, глава 8 – В.Д.Корнеевым, глава 9 – В.П.Марковой.

Книга предназначена в первую очередь для студентов и аспирантов вузов. Другая многочисленная категория читателей – начинающие прикладные программисты.

ВВЕДЕНИЕ

После долгого (по компьютерным меркам), примерно 40-летнего, развития параллельное программирование из экзотического занятия ученых довольно быстро превращается в массовую профессию. Невероятно быстрое развитие электронных технологий обеспечило появление на рынке большого числа коммерческих суперкомпьютеров. Они успешно применяются для решения многих задач в науке (математическое моделирование) и промышленности (банковские задачи, управление сложными техническими устройствами). Да и хорошо знакомые всем монопроцессорные компьютеры претерпели большие изменения и их функциональные устройства уже давно в большей или меньшей мере используют параллелизм.

Множество проблем параллельного программирования до сих пор не нашли хорошего решения в современных теориях, языках, системах и технологиях параллельного программирования. По этой причине параллельная реализация алгоритмов наталкивается на значительные трудности. Разработка, отладка и сопровождение параллельных программ оказываются весьма сложным делом. Нередко встречаются ситуации, когда долго и правильно работающая программа вдруг выдаёт ошибочные результаты.

Такие ситуации известны и в последовательном программировании. Однако в параллельном программировании

подобные проблемы усугубляются дополнительной необходимостью правильно (содержательно и во времени) запрограммировать управление и межпроцессные коммуникации, что очень трудно сделать человеку (как и любые другие действия во времени). Параллельная программа представляется системой взаимодействующих процессов, порядок выполнения которых жестко не фиксирован, что является причиной *недетерминизма* вычислений. Он проявляется, в частности, в том, что при каждом выполнении программы ее процессы исполняются, вообще говоря, в другом порядке и потому, в случае ошибки, повторить ошибочную ситуацию для локализации ошибки непросто. Каждое срабатывание неверной программы может вырабатывать другой результат. Обнаружить такие ошибки крайне сложно. По этой причине в языках параллельного программирования стараются не допускать большого недетерминизма либо ограничить его некоторыми строго определенными рамками.

Параллельные программы должны обладать необычными динамическими свойствами (настраиваемость на все доступные ресурсы, переносимость в классе мультикомпьютеров, динамическая балансировка загрузки, динамический учет свойств задачи), без которых обходятся в последовательные программы. В результате нынешнее общесистемное программное обеспечение и системы программирования оказались плохо пригодными для создания параллельных программ, непомерно

велика оказалась трудоёмкость параллельного программирования. А программы с динамическими свойствами, как известно, могут бесконечно долго отлаживаться. Хотя, конечно, есть немало задач, сравнительно просто программируемых для исполнения на мультикомпьютерах, все же многие задачи трудно распараллелить и запрограммировать. Да и как показывает практика, не так много в мире заготовлено алгоритмов, пригодных для хорошей параллельной реализации. Все это делает параллельное программирование занятием для людей, хорошо образованных в математике и хорошо обученных нужным технологиям.

Наиболее распространены сейчас системы параллельного программирования на основе MPI (Message Passing Interface). Идея MPI исходно проста и очевидна. Она предполагает представление параллельной программы в виде множества параллельно исполняющихся процессов (программы процессов обычно разработаны на обычном последовательном языке программирования C или Фортран), взаимодействующих друг с другом в ходе исполнения для передачи данных с помощью коммуникационных процедур. Они и составляют библиотеку MPI. Однако надлежащая реализация MPI для обеспечения межпроцессорных коммуникаций оказалась довольно сложной. Такая сложность связана с необходимостью достижения высокой производительности программ, необходимостью использовать

многочисленные ресурсы мультикомпьютера, и, как следствие, большим разнообразием в выборе алгоритмов реализации коммуникационных процедур в зависимости от режима обработки данных в процессорных элементах мультикомпьютера. При этом передачи данных между процессами программы должны осуществляться вне зависимости от того, где, на каких процессорах, реально находятся взаимодействующие процессы.

Другой источник сложности - учет в реализации коммуникаций особенностей передачи данных между процессами. Чем лучше учитываются особенностей оборудования и режима обработки данных в процессорных элементах, тем лучше параллельная программа. Чем меньше учет - тем более общие алгоритмы используются для реализации коммуникаций, тем хуже по времени качество передачи данных.

Программирование с использованием MPI оказалось обманчиво простым делом. Обманчивым, потому что известны простенькие очевидные параллельные программы, состоящие буквально из двух-трёх десятков операторов, которые содержат трудно обнаруживаемые ошибки. И хуже того, эти программы правильно или правдоподобно работают (особенно на тестах) и обнаруживают ошибку в самый неподходящий момент.

Для нормального освоения материала читателю необходимы знания дискретной математики, операционных систем, систем программирования, архитектуры современных

процессоров, ЭВМ и мультикомпьютеров, умение программировать, все - в рамках университетской программы факультетов прикладной математики и информатики. Следует также быть готовым к тому, что разовым кавалерийским «наскоком» материалом не овладеть. Книгу придется перечитывать и надо постараться, чтобы эти витки повторений развернулись в спираль растущего понимания предмета.

1. ПОНЯТИЕ ВЫЧИСЛИМОЙ ФУНКЦИИ

1.1. Неформальное ведение

Прежде всего, обсуждаются понятия вычислимой функции и алгоритма, которое понадобится для формирования правильной исходной позиции при анализе, конструировании и размышлениях о программах. Существует много разных формализаций понятия вычислимой функции (в [1] рассматриваются и сравниваются 12 различных формализаций понятия алгоритм и в заключение автор предлагает еще одно). Здесь обсуждается формализация Клини, следуя изложению в [2,3], которое наиболее соответствует нашим целям. За дополнительной информацией можно обратиться в [4].

Понятие вычислимой функции - ключевое понятие из тех, которыми должен владеть программист и вообще всякий, работающий в области вычислений. Задача этой главы - дать начальные понятия теории алгоритмов в ее приложении к программированию.

Говоря о программе, всегда имеют ввиду всевозможные способы ее исполнения, включая частный случай - последовательное исполнение. Термин *последовательная программа* используется, чтобы подчеркнуть единственно допустимый способ исполнения программы.

Многочисленные примеры алгоритмов известны каждому программисту. Интуитивно ясно, какие процессы могут быть отнесены к алгоритмам. Однако на интуитивном уровне понятие алгоритма остается неуловимо понятным и одновременно совершенно непонятным[1]. Каждый легко может определить, является ли некоторый предъявленный процесс алгоритмом или нет. Все, по-видимому, согласятся, что вычисление корней квадратного полинома по формуле Виета есть алгоритм. Однако без формализации понятия алгоритма невозможно показать, что не существует алгоритмического решения некоторой проблемы.

Для построения формализации понятия алгоритма можно сначала просмотреть ряд примеров алгоритмов и попробовать выделить неформальные свойства любого процесса, претендующего быть алгоритмом. Затем можно построить формальную теорию, удовлетворяющую выделенным неформальным свойствам. Так, собственно, и строится любая математическая теория.

Исходя из подобных примеров, А.И.Мальцев [2] перечисляет следующие неформальные свойства алгоритмов.

а).Алгоритм - это процесс последовательного построения величин, идущий в дискретном времени таким образом, что в

[1] В [4] авторы на стр. 13 утверждают, что "...в теории алгоритмов ... открытия состоят не столько в получении новых результатов, сколько в обнаружении новых понятий и в уточнении старых."

начальный момент задается исходная конечная система величин, а в каждый следующий момент новая конечная система величин получается по определенному закону (программе) из системы величин, имеющихся в предыдущий момент времени (*дискретность алгоритма*).

И действительно, в последовательной программе операторы исполняются, вычисляя шаг за шагом новые значения из предшествующих.

б).Система величин, полученная в какой-то (не начальный) момент времени, однозначно определяется системой величин, полученных в предшествующие моменты времени (*детерминированность алгоритма*).

Правильная программа, исполняясь несколько раз с одними и теми же входными данными, должна всегда выработать один и тот же результат.

в).Закон получения последующей системы величин из предшествующей должен быть простым и локальным (*элементарность шагов алгоритма*).

г).Если способ получения последующей величины из какой-нибудь заданной величины не дает результата, то должно быть указано, что надо считать результатом алгоритма (*направленность алгоритма*).

Если программа не может вычислить нужный результат, она должна сообщить об ошибке, описать ее и подсказать, как ее

исправить. Это и есть результат работы программы в таком случае. Но программа не должна зацикливаться без выдачи какого-либо результата! Это знает любой программист.

д).Начальная система величин может выбираться из некоторого потенциально бесконечного множества (*массовость алгоритма*).

Другое не строгое и более “машинное” описание понятия алгоритма приводит Х.Роджерс [3].

1).Алгоритм задается как конечный набор инструкций конечных размеров.

Любой алгоритм может быть описан словами, например, задан как программа на языке С.

2).Имеется вычислитель, ... который умеет обращаться с инструкциями и производить вычисления.

3).Имеются возможности для выделения, запоминания и повторения шагов вычисления.

4).Пусть P - набор инструкций из 1), L - вычислитель из 2). Тогда L взаимодействует с P так, что для любого данного входа вычисление происходит дискретным образом по шагам, без использования аналоговых устройств и соответствующих методов.

5).L взаимодействует с P так, что вычисления продвигаются вперед детерминированно, без обращения к случайным методам или устройствам.

В частности, следующий исполняемый оператор программы не может выбираться жребием.

Это неформальное определение алгоритма уточняется ответами на следующие вопросы.

6).Следует ли фиксировать конечную границу для размера входов (начальная система величин в определении Мальцева)?

Ответ нам очевиден из программирования - нет. И в самом деле, если входной величиной является файл (стандартная ситуация), его длина программе неизвестна и она работает с потенциально бесконечным входом, т.е. каждый раз обрабатывается конечная строка[2], однако для любого наперед заданного натурального N найдется строка длиной больше, чем N.

7).Следует ли фиксировать конечную границу для размера памяти?

Ответ тоже очевиден - нет. Если вычислять значение полинома второй или большей степени при неограниченной длине входа (числа символов, необходимых для представления входных значений), то ясно, что для промежуточных вычислений необходимо иметь неограниченную память (ее потребный объем зависит от длины входа). И реально программисты часто

[2] О файле говорится как о строке, потому как строкой является и любая последовательность битов, составленная, например, из записей файла

сталкиваются с ситуацией, когда программа, скажем, умножения двух матриц размера $n_0 \times n_0$ не может исполняться из-за недостатка памяти, а матрицы размера $(n_0\text{-}1)\times(n_0\text{-}1)$ умножаются благополучно. Можно задавать и другие вопросы, но для нас этого достаточно.

Этих не строгих определений на практике достаточно и для того, чтобы понять, является ли процесс алгоритмом, и для построения нужного алгоритма. Однако его совершенно недостаточно для ответа на вопрос, существует ли алгоритм решения некоторой проблемы? Для доказательства существования алгоритма решения проблемы достаточно построить такой алгоритм. Здесь годится и неформальное определение. Но чтобы доказать отсутствие такого алгоритма, надо для начала точно знать, что такое алгоритм. Нам строгое определения алгоритма поможет определить необходимое понятие программы и сформулировать требования к ней.

Каждый алгоритм A вычисляет функцию F_A при некоторых заданных значениях входных величин. Функции, вычисляемые алгоритмом, называются *алгоритмически вычислимыми* функциями. Понятие вычислимой функции, так же как и понятие алгоритма, тоже здесь интуитивно. Существует много разных формализаций понятия алгоритм, удовлетворяющих неформальным требованиям. Однако класс алгоритмически вычислимых функций, определенный во всех

таких формализациях оказался одним и тем же при самых разнообразных попытках расширить понятие алгоритма.

Черч высказал гипотезу - *тезис Черча*, что класс всех вычислимых функций совпадает с классом алгоритмически вычислимых функций. Так как понятие вычислимой функции определено неформально, то и тезис Черча строго недоказуем. Если принять тезис Черча, то тогда доказательство вычислимости функции сводится к доказательству ее алгоритмической вычислимости.

Пост и Тьюринг для уточнения понятия алгоритма построили точно описанные в математических терминах “машины”. Несмотря на предельную тривиальность этих машин на них оказалось возможным определить (выполнить) все алгоритмические процессы, когда-либо рассматриваемые в математике.

1.2. Основные предварительные понятия

Задача этого раздела - обсудить ряд необходимых понятий и свести рассмотрение к классу только числовых функций, после чего можно будет вплотную заняться формализацией понятия алгоритмически вычислимой функции.

1.2.1. Алфавит.

Алфавитом называется конечное множество символов $A=(a,b,...,c)$. Конечная последовательность символов алфавита A называется *словом* в алфавите A. Количество символов в слове называется *длиной* слова. Например, если $A=\{a,b\}$, то ab, aa, $abbaa$, $bbbbba$ - слова в алфавите A. Множество всех слов алфавита A обозначается A^*.

Если s_1 и s_2 - слова, то слово s_1s_2 называется *конкатенацией* (*произведением*) слов s_1 и s_2. *Пустое слово* не содержит символов, его конкатенация с любым словом s равна слову s. Слово s_2 называется *подсловом* слова s, если s представимо в виде $s = s_1s_2s_3$, s_1 и s_3 - тоже слова в алфавите A, возможно пустые. Слово $s_1s_2s_3$ называется *разложением* слова s, s_1 - *префикс* разложения $s_1s_2s_3$.

Возможно несколько различных разложений слова s, содержащих подслово s_2. Слово s_2 называется первым *вхождением* s_2 в s, если префикс s_1 обладает наименьшей длиной среди всех таких разложений слова s. Например, слово $abaabaaba=a\cdot ba\cdot abaaba=abaa\cdot ba\cdot aba=abaabaa\cdot ba$ имеет несколько разных разложений с подсловом ba, первое из них содержит первое вхождение ba в слово $abaabaaba$.

1.2.2. Кодирование.

В теории всегда рассматриваются вычислимые функции, отображающие одни множества слов в другие. Множества слов в некотором алфавите выбраны в силу универсальности этих множеств для представления величин, потому что все прочие виды величин кодируются и представляются словами.

Пусть A - алфавит, A^* - множество всех слов в алфавите A, $B \subseteq A^*$, M - множество некоторых объектов. Закодировать объекты из M в алфавите A означает задать однозначное соответствие $f: B \to M$. Взаимно однозначного соответствия не требуется. Слово $b \in B$, $m=f(b)$, $m \in M$, называется *кодом* или *именем* объекта m в кодировании f.

Рассмотрим алфавит $A=\{1\}$, тогда натуральное число n может быть представлено словом 111...1, содержащем n единиц, а само слово 111...1 называется *кодом* числа n. Если использовать алфавит $A=\{0,1\}$, то всем программистам известно, как кодируются целые и вещественные числа, а также символы различных алфавитов. Слово 11 есть код числа 3, а слово 011, вообще говоря, не является кодом никакого числа, если специально не оговорить случай незначащих предшествующих нулей (что и сделано во всех компьютерах). Таким образом, при кодировании возможны несколько различных кодов одной и той же величины и возможны коды, не представляющие никакой

величины. Однако всегда по заданному коду закодированный объект должен восстанавливаться однозначно.

Понятно, что при изучении вычислимых функций можно ограничиться рассмотрением только *числовых* функций (функций, определенных на множестве натуральных чисел $\boldsymbol{N}$, со значениями в $\boldsymbol{N}$), предполагая наличие подходящей кодировки. И все рассматриваемые множества – только подмножества множества натуральных чисел $\boldsymbol{N}$ либо само $\boldsymbol{N}$.

1.2.3. Бесконечный алфавит

Иногда удобно использовать бесконечные алфавиты вида $A=\{a,b,...,c,d_i,t_i^j\}$, содержащий конечное число символов $a,b,...,c$, $i,j=0,1,...$. Такой алфавит формально бесконечен, так как содержит бесконечное число символов d_i, t_i^j. Однако он легко кодируется в конечном алфавите $A_0=\{a,b,...,c,d,t,n,m\}$. При этом символы d_i кодируются словами $dnn...n$, а символы t_i^j - словами $tnn...nmm...m$, в которых символ n повторяется i раз, а символ m - j раз. Таким образом, всегда можно оставаться в конечных алфавитах даже при использовании такого сорта бесконечных алфавитов.

1.2.4. Наборы (кортежи)

Пусть задан алфавит, $B=A\cup\{,\}$, где "," - специальный выделенный символ. Кроме слов в A придется далее

рассматривать наборы слов вида $<s_1,s_2>$ $<s_1,s_2,s_3>$ и т.д. Такие наборы слов алфавита A кодируются словами в алфавите B, при этом слово s_1s_2 кодируется словом $<s_1,s_2>$, а слово $s_1s_2s_3$ - словом $<s_1,s_2,s_3>$.

1.2.5. Термы

Термы (*функциональные термы*) - это слова особого вида, записанные в функциональном алфавите. *Функциональный алфавит* состоит из трех групп символов, используемых для записи функций. Первую группу составляют *предметные* символы (*предметные переменные*), изображающие переменные. Для их обозначения будут использованы строчные буквы *x, y, z,* или эти же символы с индексами. Вторую группу составляют *функциональные* символы, изображающие функции. В качестве функциональных символов будем использовать буквы *f, g, h* или эти же символы с верхними и нижними индексами. Верхний индекс обычно указывает арность (местность) функционального символа. Третью группу составляют специальные символы скобок " (", ") " и запятая " , ".

Понятие терма определяется шагами по индукции. Сначала определяются термы длины 1, т.е. слова длины 1 в функциональном алфавите, которые есть термы по определению. Термами длины 1 называются односимвольные слова из

предметных символов. Например, термами являются слова x и y, а слово f термом не является.

Далее, пусть для некоторого, $k>1$, термы длины меньше k уже определены. Тогда слово t длины k называется термом, если оно имеет вид $f(t_1,t_2,...,t_m)$, где $t_1, t_2, ..., t_m$ - термы длины меньшей чем k, f – *m-арный* функциональный символ (m входных переменных). Пусть, например, дан функциональный алфавит $A = \{x, y, a\} \cup \{f, g^2\} \cup \{(,), ,\}$. Тогда слова $f(x)$, $g^2(y,a)$ $g^2(f(x), g^2(y,a))$ есть термы. На рис. 1.1 показано графическое представление последнего терма.

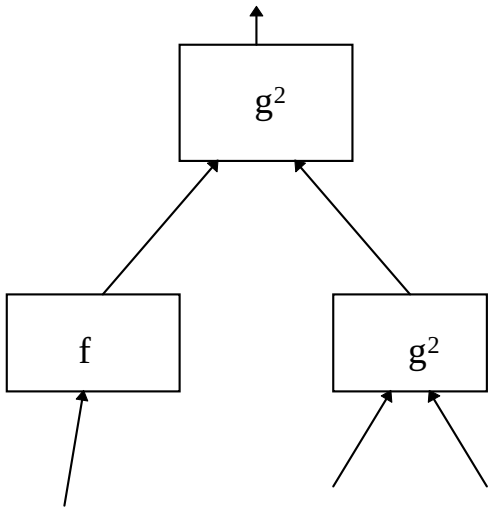


Рис. 1.1.

Понятие терма введено чисто синтаксически безотносительно к понятиям функции, переменной и т.п. Просто есть алфавит, символы которого разбиты на три подмножества и слова особого вида названы термами. Сейчас будет введено понятие интерпретации, которое только и придаст смысл терминам «предметный символ» и «функциональный символ».

Интерпретацией $\boldsymbol{I}$ называется отображение, которое сопоставляет:

- каждой предметной переменной x - элемент $\boldsymbol{I}(x)=d_x\in\boldsymbol{N}$, элемент d_x множества $\boldsymbol{N}$ называется *значением переменной* x, $\boldsymbol{N}$ называется *осно́вным* множеством или *областью интерпретации*.

- каждому k-местному функциональному символу f^k - частичную функцию $\boldsymbol{I}(f^k)=F^k:\boldsymbol{N}\times\boldsymbol{N}\times\ldots\times\boldsymbol{N}\to\boldsymbol{N}$. Функция F^k называется *значением функционального символа* f^k.

- каждому терму $t=f^k(x1,x2,\ldots,xk)$ сопоставляется значение терма $\boldsymbol{I}(t)=d_t\in\boldsymbol{N}$, $d_t=val(f^k(x1,x2,\ldots,xk))$. Функция *val* (от английского *value* - значение) задана на множестве термов, она определяется по индукции:

- $val(x)=\boldsymbol{I}(x)=d_x\in D_x$;

- пусть $t=f^k(t1,t2,\ldots,tk)$ - терм длины n, термы $t1$, $t2$, ..., tk длины меньшей, чем n и функция *val* определена для термов длины меньшей чем n.

Тогда $val(t)=val(f^k(t1,t2,...,tk))=F^k(val(t1),(val(tk),...,(val(tk))$, $F^k=\boldsymbol{I}(f^k)$ - значение функционального символа f^k.

Таким образом, в качестве значения терма $t=f^k(t1, t2, ... , tk)$ ему сопоставляется то значение функции F^k, которое она имеет при значениях ее аргументов, равных $val(t1), val(t2), ... , val(tk)$.

Например, если $\boldsymbol{I}(f)=F$ и $\boldsymbol{I}(g)=G^2$, тогда

$val(x)=d_x$, $val(y)=d_y$, $val(f(x))=F(val(x))=F(d_x)$,

$val(g(x, y))=G^2(val(x), val(y))=G^2(d_x, d_y)$,

$val(g(f(x),g(x,y))=G^2(val(f(x),val(g(x,y)))=$

$G^2(F(val(x)), G^2(val(x),val(y)))=G^2(F(d_x), G^2(d_x, d_y))$.

Понятно, что функциональный терм задает новую функцию, определенную как комбинацию заданных функций. Если эти функции частичны, то и терм определяет частичную функцию.

Такого определения понятия терм достаточно в этой главе. Однако для целей программирования оно будет далее уточняться.

1.3.Понятие рекурсивной функции

Теперь можно приступить к определению понятия рекурсивной (алгоритмически вычислимой) функции. Идея определения состоит в том, чтобы сначала выбрать множество простейших функций, и сказать, что они вычислимы “по

определению". Простейшие функции выбираются таким образом, чтобы в их вычислимости не было никаких сомнений.

Затем определяются *операторы* - схемы конструирования новых вычислимых функций из имеющихся. Каждый оператор будет определен таким образом, что в его вычислимости тоже не возникнет сомнений. Применение этих операторов к вычислимым функциям вновь должно дать вычислимую функцию. Необходимо так выбрать операторы, чтобы с их помощью (конечной комбинацией их применений) можно было построить любую вычислимую функцию.

1.3.1. Простейшие вычислимые функции

Итак, на первом шаге выбирается счетный *базис* простейших *рекурсивных* функций, вычислимых по определению. Это следующие функции:

1. Функция следования $s(x)=x+1$
2. Нулевая функция $o(x)=0$
3. Функции выбора $I_m^n(x_1,x_2,\ldots,x_n)=x_m,\ m<n$

Функции устроены крайне просто. Ни у кого, видимо, нет сомнений в интуитивной вычислимости этих функций, в возможности построить некоторое "механическое" устройство, вычисляющее эти функции.

1.3.2. Суперпозиция вычислимых функций

Пусть заданы n частичных функций m переменных $f_i^m: D_1 \times D_2 \times ... \times D_m \to D_o$, $i=1, 2, ..., n$, и пусть определена функция $f^n: D_0 \times D_0 \times ... \times D_0 \to D$[3]. Определим функцию m переменных g^m, определенную на множестве $D_1 \times D_2 \times ... \times D_m$ со значениями в D, $g^m(x_1,x_2,...,x_m)=f^n(f_1^m(x_1,x_2,...,x_m),...,f_n^m(x_1,x_2,...,x_m))$. Функция g^m конструируется *суперпозицией* (*подстановкой*) из функций f^n, f_1^m, ... , f_n^m. Оператор суперпозиции будем обозначать $\mathbf{S^{n+1}}$. Например, функция $\mathbf{S^3}(I_1^2(x_1,x_2), I_3^4(x_1,x_2,x_3,x_4), I_2^4(x_1,x_2,x_3,x_4))=I_1^4(x_1,x_2,x_3,x_4)$.

Понятно, что функция $g^m(x_1,x_2,...,x_m)$ и есть та функция, которую определяет функциональный терм $f^n(f_1^m(x_1,x_2,...,x_m),...,f_n^m(x_1,x_2,...,x_m))$ (рис. 1.2) при соответствующей интерпретации.

[3] Напомним, что в качестве областей определения и значения функций в главе всегда берутся либо множество натуральных чисел $\boldsymbol{N}$ либо его подмножества. Следовательно, $D, D_o, D_1, D_2, ... D_m \subseteq \boldsymbol{N}$.

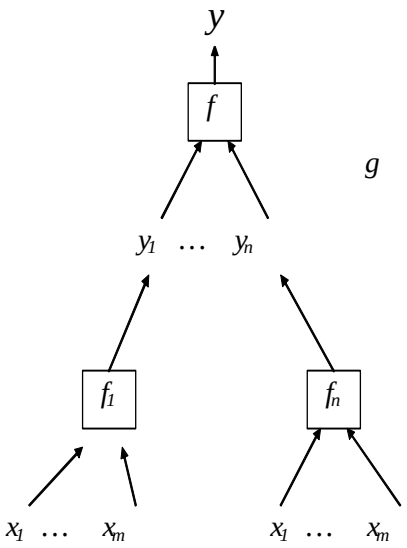


Рис. 1.2.

Таким образом, функция $g=\mathbf{S}^{n+1}(f,f_1,...,f_n)$ определяется (вычисляется, задается) функциональным термом (рис.1.2). Переменные y_1, y_2, ... ,y_m представляют промежуточные величины.

Пусть ϕ^n - множество всех частичных вычислимых функций от n переменных. Тогда оператор суперпозиции $\mathbf{S}^{n+1}$ может рассматриваться как всюду определенная функция $\mathbf{S}^{n+1}:\phi^n\times\phi^m\times\phi^m\times \ ... \ \times\phi^m \rightarrow \phi^m$. Теперь в этих терминах можно перефразировать понятие функции, полученной применением оператора суперпозиции, а именно:

функция $g(x_1,x_2,...,x_m)$ получается суперпозицией из функций $f(x_1,x_2,...,x_n)$, $f_1(x_1,x_2,...,x_m)$, ... , $f_n(x_1,x_2,...,x_m)$, если g есть значение *операторного* терма $\mathbf{S^{n+1}}(h,h_1,...,h_n)$, где h, h_1, ... , h_n - переменные (здесь функциональные символы) и интерпретация сопоставляет:

переменной h - значение f, элемент из ϕ^n,

переменным h_1, ... , h_n - значения f_i из ϕ^m соответственно, i=1, 2, ..., n,

функциональному символу $\mathbf{S^{n+1}}$ - функцию $f(f_1(x_1,x_2,...,x_m),... ,f_n(x_1,x_2,...,x_m)=g(x_1,x_2,...,x_m)$.

Таким образом, можно использовать либо операторную либо термальную запись представления функции g. В первой из них g есть значение операторного терма, во второй g определяется как функциональный терм[4].

Например, значением операторного терма $\mathbf{S^3}(I_1^2(x_1,x_2), I_3^4(x_1,x_2,x_3,x_4), I_2^4(x_1,x_2,x_3,x_4))$ есть функция $I_1^4(x_1,x_2,x_3,x_4)$ (здесь в операторный терм уже подставлены значения $I_1^2(x_1, x_2)$, $I_3^4(x_1, x_2, x_3, x_4)$, $I_2^4(x_1, x_2, x_3, x_4)$ переменных.

Возьмем функцию $x_1(x_2+x_3)$ в термальном представлении, в качестве символов функций использованы символы функций

[4] Этот функциональный терм и определяет алгоритм вычисления функции g.

сложения + и умножения ×. Ее операторное представление ест $\mathbf{S}^3(\times, I_1^3, \mathbf{S}^3(+, I_2^3, I_3^3))$.

Программисты легко могут представить себе наличие процедур, вычисляющих простейшие вычислимые функции. Оператор суперпозиции определяет, с позиций программирования, простое “сцепление” процедур, при котором одни процедуры вырабатывают значения своих выходных переменных, а другие их использует в качестве входных величин. Понятен и “механический” характер определения оператора суперпозиции, его очевидная вычислимость, а именно: если известно, как вычислить значения функций $f, f_1, \ldots, f_n$, то понятен и алгоритм вычисления функции $g(x_1,x_2,\ldots,x_m)=f(f_1(x_1,x_2,\ldots,x_m), \ldots,f_n(x_1,x_2,\ldots,x_m))$.

Каждый программист легко узнает в нижеследующей программе “реализацию” оператора суперпозиции. Слово “реализация” взято в кавычки, чтобы подчеркнуть его неформальность. Позднее этот термин будет точно определен.

Пусть процедуры $f, f_1, \ldots, f_n$ вычисляют одноименные функции. Тогда программа P вычисляет функцию g.

P: $y_1 := f_1(x_1,x_2,\ldots,x_m)$;

. . .

$y_n := f_n(x_1,x_2,\ldots,x_m))$

$y := f(y_1,y_2,\ldots, y_n)$;

Если применить оператор суперпозиции конечное число раз, то будет построено конечное число функциональных термов, а значит, конечным числом применений оператора суперпозиции к простейшим функциям могут быть построены только вычислимые функции. Понятно, что применением оператора суперпозиции к простейшим функциям (других нет ещё сейчас) все алгоритмически вычислимые функции построить не удастся. Значит надо определять другой, более мощный оператор.

1.3.3. Оператор примитивной рекурсии

Оператор примитивной рекурсии устроен более сложно, чем оператор суперпозиции. Он позволяет определить циклические вычисления специального вида.

Пусть заданы:

- n-местная вычислимая функция g и
- $(n+2)$-местная вычислимая функция h.

Тогда $(n+1)$-местная функция f строится *примитивной рекурсией* из функций g и h, если для всех натуральных значений $x_1, x_2, \ldots, x_n, y$ имеем:

$$f(x_1,x_2,\ldots,x_n,0)=g(x_1,x_2,\ldots,x_n)$$
$$f(x_1,x_2,\ldots,x_n,y+1)=h(x_1,x_2,\ldots,x_n,y,f(x_1,x_2,\ldots,x_n,y)) \quad (1)$$

Если $n=0$, то одноместная функция f строится примитивной рекурсией из функции-константы $Cnst$ (это просто некоторое натуральное число) и двухместной функции h

$f(0)=Cnst$, $f(x+1)=h(x,f(x))$

Соотношения (1) называются *схемой примитивной рекурсии*. Они определяют *оператор примитивной рекурсии* **R**. Эта схема, собственно, и есть алгоритм вычисления функции *f*. Оператор **R** определен на множестве ϕ частичных вычислимых функций, $f=\mathbf{R}(g,h)$.

Понятно, что функция f существует и единственна. Это следует из того, что определение функции, построенной примитивной рекурсией, фактически содержит схему (алгоритм) ее вычисления. Распишем детально эту схему, используя термальные представления.

Пусть необходимо вычислить значение функции f при $y=k$. Тогда из определения (1) схемы примитивной рекурсии имеем последовательность функциональных термов t_0, t_1,…, t_k, вычисляющих значение функции $f(x_1, x_2, \ldots, x_n, k)$:

t_0: $f_0=f(x_1,x_2,\ldots,x_n,0)=g(x_1,x_2,\ldots,x_n)$

t_1: $f_1=f(x_1,x_2,\ldots,x_n,1)=h(x_1,x_2,\ldots,x_n,0,g(x_1,x_2,\ldots,x_n))$

t_2: $f_2=f(x_1,x_2,\ldots,x_n,2)=h(x_1,x_2,\ldots,x_n,1,f(x_1,x_2,\ldots,x_n,1))$

.....

t_k: $f_k=f(x_1,x_2,\ldots,x_n,k)=h(x_1,x_2,\ldots,x_n,k-1,f(x_1,x_2,\ldots,x_n,k-1))$

Здесь представлен способ вычисления *f*, следовательно, функция f определена однозначно. Если одна из функций $f(x_1,x_2,\ldots,x_n,m)$, $m<k$, не определена, то и значение $f(x_1,x_2,\ldots,x_n,y)$ для всех $y\geq m$ тоже не определено. Эту последовательность $k+1$

функциональных термов t_0, t_1, ... t_k удобно для наглядности изобразить графически (рис. 1.3).

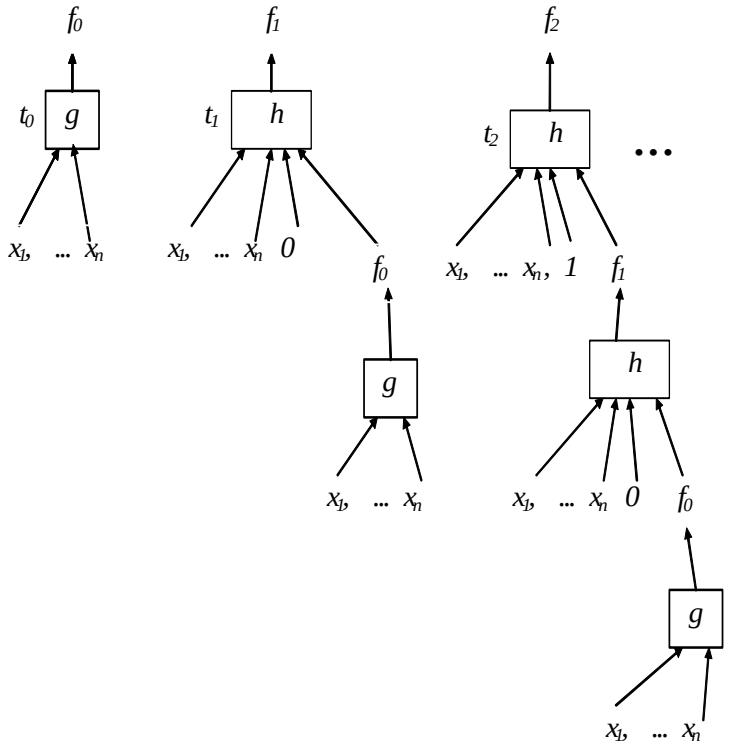


Рис. 1.3.

Это множество функциональных термов и определяет алгоритм вычисления функции *f*. Опытные программисты легко узнают на рис. 1.3. «раскрутку» вызовов рекурсивной процедуры. Множество термов (представление алгоритмов в виде множества термов) позволяет легко построить программу *P*, реализующую

применение оператора примитивной рекурсии к функциям g и h и вычисляющую функцию f.

P: *s*:=*k*;

f[0]:=*g*(x_1,x_2,...,x_n);

for *i*=1 **to** *s* **do**

f[i]=h(x_1,x_2,...,x_n,*i-1*,*f[i-1]*);

f :=*f[s]*;

Программисты записывают, обычно, *P* более экономно, но не "функционально".

P1 : *s*:=*k*;

f:=*g*(x_1,x_2,...,x_n);

for *i=1* **to** *s* **do**

f=h(x_1,x_2,...,x_n,*i-1*,*f*);

Отметим следующие важные обстоятельства этого определения:

1.Если мы умеем вычислять функции g и h, то значение функции $f(x_1,x_2,...,x_n,k)$ вычисляется опять же "механически", т.е. алгоритмом. Таким образом, следует признать функцию f вычислимой.

2.Для любого натурального числа k значение функции $f(x_1,x_2,...,x_n,k)$ задается/вычисляется k функциональными термами (рис.1.3).

3.Так как число k - любое натуральное число, то функция f, полученная применением оператора примитивной рекурсии к функциям g и h, может быть определена потенциально

бесконечным множеством функциональных термов (рис. 1.3) особого вида. Перефразируя, можно сказать, что функция f может быть построена потенциально бесконечным применением оператора суперпозиции.

4.В программе оператор примитивной рекурсии реализуется циклом типа **for**[5]. Граница изменения параметра цикла (число k в программе P) должна быть определена до начала выполнения цикла. Это характерное свойство примитивно рекурсивных функций: значения входных величин не ограничены, однако они должны принять конкретное конечное значение до начала исполнения цикла.

Функция f называется *примитивно рекурсивной* относительно множества простейших функций, если она строится из простейших вычислимых функций конечным числом применения операторов суперпозиции и примитивной рекурсии.

Если расширить множество простейших функций добавлением в него множества всюду определенных примитивно рекурсивных функций, то применяя к функциям расширенного множества конечное число раз операторы примитивной рекурсии и суперпозиции будут вновь построены примитивно рекурсивные функции. Понятно, что все примитивно рекурсивные функции всюду определены.

[5] По этой причине в любом процедурном языке программирования должен оператор типа **for** (**DO** в Фортране, например)

П1.По определению функции $s(x)=x+1$, $o^1(x)=o(x)=0$, $I_m^n(x_1,x_2,...,x_n)=x_m$, $m<n$, являются примитивно рекурсивными функциями. Функция $o^n(x_1,x_2,...,x_n)=0$ представляется операторным термом $o^n=\mathbf{S^2}(o^1,I_1^n)$**.** Следовательно, функция $o^n(x_1,x_2,...,x_n)=0$ примитивно рекурсивна.

П2. Рассмотрим примитивно рекурсивную схему

$x+0=x=I_1^1(x)$

$x+(y+1)=(x+y)+1=s(x+y)$

Следовательно, функция $(x+y)$ строится применением оператора примитивной рекурсии **R** к примитивно рекурсивным функциям $g(x)=I_1^1$ и $h(x,y,z)=z+1$. Следовательно, функция $x+y$ примитивно рекурсивна.

П3. Аналогично, функция xy удовлетворяет следующей схеме примитивной рекурсии

$x\cdot 0=o(x)$

$x\cdot(y+1)=xy+x$

с $o(x)$ в качестве функции g и $h(x,y,z)=z+x$. Следовательно, функция xy примитивно рекурсивна.

Следует различать понятия алгоритмически вычислимой функции и алгоритма, а именно: пусть существует некоторая нерешенная математическая проблема. Определим функцию

$$h(x)=\begin{cases}1, \text{если проблема решается положительно}\\ 0, \text{если проблема решается отрицательно}\end{cases}$$

Ясно, что функция $h(x)$-константа (либо 0, либо 1) и, следовательно, примитивно рекурсивна. Однако алгоритм (примитивно рекурсивная схема) ее вычисления неизвестен, поскольку функции $h(x)=0$ и $h(x)=1$ вычисляются разными алгоритмами, а сделать правильный выбор невозможно (нет решения проблемы).

Как оказалось, класс примитивно рекурсивных функций не исчерпывает всех вычислимых функций. Нашлись с очевидностью алгоритмически вычислимые функции, для которых показано не существование примитивно рекурсивных схем. Одна из таких функций - функция Аккермана.

$f(0,x,y)=y+x$

$f(1,x,y)=yx$

$f(2,x,y)=y^x$

. . . . .

Таким образом, необходимо расширить класс примитивно рекурсивных функций в попытке определить класс всех вычислимых функций.

Насколько, однако, далеко можно строить расширения? Попробуем в этом разобраться. Прежде всего заметим, что для определения класса примитивно рекурсивных функций использовалось конечное число символов: символы простейших функций, вспомогательные символы типа скобок “(“ и “)”, запятой “,”, равенства =, символ конца строки.

Далее, всякая примитивно рекурсивная функция строится конечным числом применения операторов суперпозиции и примитивной рекурсии. Следовательно, определение каждой примитивно рекурсивной функции задается в выбранном формализме конечной строкой символов.

Если задана конечная строка символов, то понятно, что можно построить алгоритмическую процедуру для распознавания, определяет ли строка схему (1) примитивной рекурсии или нет.

Все примитивно рекурсивные схемы могут быть последовательно перечислены: сначала анализируются все строки длины 1 (их конечное число) и отбираются те, что определяют примитивно рекурсивные схемы. Затем так же исследуются все строки длины 2 и т.д. Каждое множество строк длины k конечно (оно может быть упорядочено произвольным образом, например, лексикографически) и для каждого k такой анализ закончится за конечное время (за конечное число шагов). При анализе распознанные примитивно рекурсивные схемы последовательно нумеруются.

Пусть теперь G_x есть схема с номером x, а g_x - примитивно рекурсивная функция, определяемая этой схемой. По аналогии с диагональным методом Кантора определим функцию h:

$$h(x)=g_x(x)+1$$

Ясно, что функция $h(x)$ вычислимая. Для любого натурального x в качестве значения функции $h(x)$ берется значение функции $g_x(x)$, вычисляющейся схемой G_x, плюс 1. Ясно также, что функция $h(x)$ не примитивно рекурсивна. Если предположить ее примитивную рекурсивность, то тогда существует такое число $x0$, что $h(x0)=g_{x0}(x0)$, т.е. при перечислении будет выбрана схема G_{x0}, определяющая функцию $h(x)$. При этом из определения функции $h(x)$ сразу возникает противоречие

$$h(x0)=g_{x0}(x0)=g_{x0}(x0)+1$$

Попытки расширить формально определяемый класс вычислимых функций выглядят теперь бесперспективными. К каждому вновь определенному классу вычислимых функций можно будет применить описанную выше процедуру и показать, что вне его есть другие вычислимые функции, а следовательно, требуется новое расширение класса.

Однако проведенные рассмотрения дают намек, как можно избежать противоречия (2), а именно: новый оператор следует определить таким образом, чтобы он задавал частичные функции. В этом случае функция $g_{x0}(x)$, а с нею и функция $h(x)$, не обязательно должны быть вычислимыми при $x=x0$! Возможно, что вычисление их значения в $x=x0$ никогда не завершается и это значение останется неопределенным, а значит

и нет противоречия в (2). Именно так сконструирован оператор минимизации.

1.3.4. Оператор минимизации

Пусть f - некоторая n-местная вычислимая функция, $n \geq 1$, алгоритм вычисления f известен. Вычисление значения функции f для некоторого значения аргумента невозможно лишь тогда, когда алгоритм не может завершиться.

Зафиксируем некоторые значения первых n-1 переменных $x_1, x_2, \ldots, x_{n-1}$. Рассмотрим уравнение

$$f(x_1,x_2,\ldots,x_{n-1},y)=x_n \quad (3)$$

Для его решения начнем вычислять последовательно значения функции $f(x_1,x_2,\ldots,x_{n-1},y)$ для y=0, 1, 2, ... Наименьшее число k такое, что $f(x_1,x_2,\ldots,x_{n-1},k)=x_n$ есть решение этого уравнения. Это решение обозначается $\mu_y(f(x_1,x_2,\ldots,x_{n-1},y)=x_n)$. Решения может и не оказаться, например если:

- значения $f(x_1,x_2,\ldots,x_{n-1},k)$, $k=0, 1, 2, \ldots$ определены и отличны от x_n, а значение $f(x_1,x_2,\ldots,x_{n-1},k+1)$ не определено,

- значения $f(x_1,x_2,\ldots,x_{n-1},k)$, $k=0, 1, 2, \ldots$, все определены и отличны от x_n.

В таких случаях значение $\mu_y(f(x_1,x_2,\ldots,x_{n-1},y)=x_n)$ считается неопределенным. Величина $\mu_y(f(x_1,x_2,\ldots,x_{n-1},y)=x_n)$ зависит от значений переменных $x_1, x_2, \ldots, x_{n-1}, x_n$ и потому она может

рассматриваться как значение функции от аргументов x_1, x_2, ... ,x_{n-1}, x_n. Эта функция обозначается $\boldsymbol{M}f$, $\boldsymbol{M}$ называется оператором *минимизации*. Если f - одноместная функция, то очевидно, что $f^{-1}(x)=\mu_y(f(y)=x)$.

Оператор минимизации очевидно частично вычислим, а значит и функция $\boldsymbol{M}f$ частично вычислима. Функция f называется *частично рекурсивной* относительно простейших функций, если она строится конечным числом применений операторов суперпозиции, примитивной рекурсии и минимизации.

Перефразируя, можно сказать, что функция называется частично рекурсивной, если она представляется конечным операторным термом. В соответствии с тезисом Черча класс частично-рекурсивных функций совпадает с классом всех вычислимых функций.

Как и для оператора примитивной рекурсии для оператора минимизации можно построить представление в виде счетного множества функциональных термов.

Введем предикат $P(y_i)$:$f(x_1,x_2,...,x_{n-1},y_i)=x_n$. Тогда оператор минимизации представляется счетным множеством функциональных термов (рис. 1.4). Предикат $P(y_i)$ выделяет функциональный терм, который вырабатывает значение функции $\mu_y(f(x_1,x_2,...,x_{n-1},y)=x_n)$.

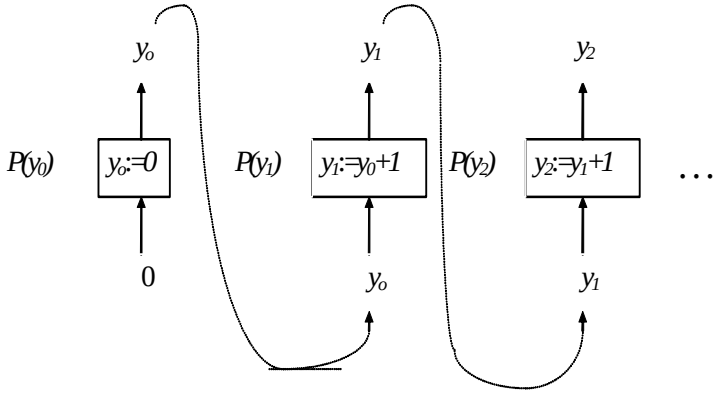


Рис. 1.4.

Из всех этих термов лишь один может выработать значение функции $\boldsymbol{M}f$, а именно тот, который выработает значение $z_k=x_n$ для минимального k. Таким образом и здесь, как и в случае оператора примитивной рекурсии, операторный терм $\boldsymbol{M}f$ представляется счетным множеством функциональных термов. Одно из существенных отличий состоит в том, что конечное множество функциональных термов, представляющих оператор примитивной рекурсии, фиксируется для конкретных значений до начала вычислений. В случае оператора минимизации терм, вычисляющий значение функции, определяется динамически в ходе вычисления.

Для конструирования программы, вычисляющей функцию на рис. 1.4. оператора типа **for** уже недостаточно, нужен оператор типа **while**. Именно поэтому, любой

процедурный язык программирования содержит операторы типа **while**. Следующая программа реализует оператор минимизации:

P : $z:=f(x_1,x_2,...,x_{n-1},0)$;

$i:=0$;

while $z \neq x_n$ **do**

{

$i:=i+1$;

$z:=f(x_1,x_2,...,x_{n-1},i)$;

};

$Mf:=i$;

Понятно, что эта программа не всегда останавливается и, следовательно, вычисляемая ею функция может оказаться не всюду определенной.

1.4.Детерминант вычислимой функции

Следуя [1] определим понятие вычислимой функции, которое более удобно для работы с параллельными алгоритмами и программами.

Характеристической функцией X_A множества A называется одноместная функция, равная 0 на элементах множества A и 1 за пределами A. Характеристическая функция называется *частичной*, если она не определена за пределами A. Множество A называется примитивно рекурсивным, если его характеристическая функция примитивно рекурсивна.

Множество *A* называется частично рекурсивным, если его характеристическая функция частично рекурсивна.

Множество *A* называется *рекурсивно перечислимым*, если существует двухместная частично рекурсивная функция *f(a,x)* такая, что уравнение *f(a,x)*=0 имеет решение тогда и только тогда, когда $a \in A$.

Было показано, что каждая частично рекурсивная функция представляется конечным операторным термом. Этому представлению соответствует представление частично рекурсивной функции потенциально бесконечным множеством функциональных термов. Это множество, очевидно, рекурсивно перечислимое, оно устроено специальным образом (для каждого оператора свое множество функциональных термов) и существует алгоритм, перечисляющий термы множества. Для каждого операторного терма рассмотрены примеры программ, которые реализуют представляющее его множество функциональных термов.

И наоборот, каждой программе может быть сопоставлено множество функциональных термов, определяющее тот алгоритм, что реализован программой. Например, программа покомпонентного сложения двух векторов

```
n:=.... ;
for  i = 1  to  n  do
        z[i]:=x[i]+y[i] ;
```

реализует алгоритм, представленный множеством термов

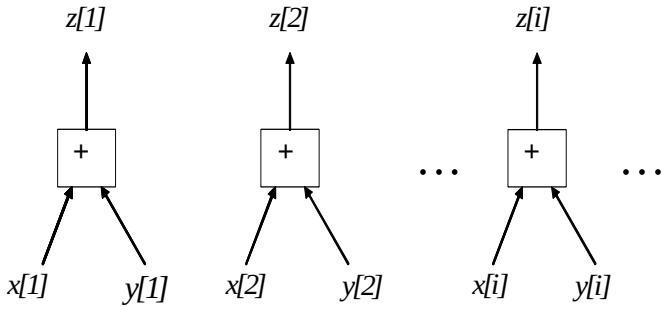


Рис. 1.5.

Рассмотрим другой пример.

n:=.... ;

for *i = 1* **to** *n* **do**

if *P(x*[*i*])*>0* **then** *z*[*i*]:=*x*[*i*]+*y*[*i*] **else** *z*[*i*]:=*x*[*i*]+*y*[*i*] ;

Реализованный программой алгоритм может быть представлен множеством функциональных термов

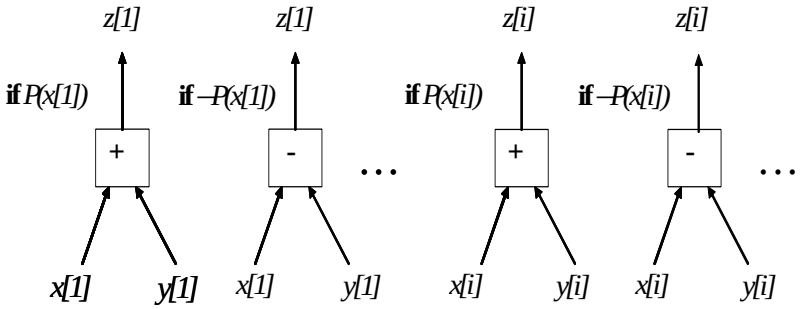


Рис. 1.6.

Это множество функциональных термов строится фиксацией значения предиката *P*. Если фиксировать *P*=**true**, то программа реализует множество функциональных термов с использованием только операции “**+**”. А если зафиксировать *P*=**false**, то программа реализует множество функциональных термов с операцией “**-**“ . Объединение этих множеств и даст множество, показанное на рис. 1.6.

Замечание. В разделе 1.2.5. были введены понятия терма и интерпретации ***I***. Интерпретация ***I*** определяет, каким образом присваиваются значения всем предметным и функциональным символам алгоритма. Если отвлечься от несущественных в этом рассмотрении деталей, то компьютер воспринимает программу именно как множество функциональных термов, а заложенный в его оборудовании алгоритм вычисляет интерпретацию ***I***, т.е. присваивает значения всем предметным и функциональным символам алгоритма. Именно вычисление ***I*** и есть функция компьютера.

Определение. Функция $f:D^k \to D$ называется *вычислимой функцией*, если существует конечно порожденное множество T функциональных термов и выполняется:

1. $\forall(x_1,x_2,...,x_k,y)$ таких, что $y=f(x_1,x_2,...,x_k,)$ $\exists t(x_1,x_2,...,x_k)\in T$ такой, что $d_y=val(t)=val(t(x_1,x_2,...,x_k))$;

2. $\forall(x_1,x_2,...,x_k,y)\ \forall t(x_1,x_2,...,x_k)\in T$, если $val(t(x_1,x_2,...,x_k))=d_y$ то $y=f(x_1,x_2,...,x_k,y)$, т.е. T не содержит “лишних” термов, не вычисляющих функцию f.

Множество термов T называется *детерминантом* Det_f функции f []. Детерминант Det_f определяет алгоритм вычисления функции f. Если заданы два различных детерминанта $Det1^1_f$ и $Det1^2_f$ (существует счетное множество алгоритмов, вычисляющих f), то $Det1^1_f \cup Det1^2_f$ тоже является детерминантом. Ясно, что детерминант может содержать два различных терма, вычисляющих значение одной и той же переменной, и, следовательно, многозначность функции f не исключается в общем случае.

Процедурные языки программирования типа С и Фортран содержат именно средства для реализации операторов суперпозиции, примитивной рекурсии и минимизации. Все прочие "излишества" языков (типа оператора **go to** в языке С, блок **COMMON** в Фортране и т.п.) служат для упрощения записи алгоритмов и улучшения качества их реализации и непринципиальны с точки зрения теории. В этом смысле все процедурные языки программирования похожи друг на друга как близнецы-братья.

Рассмотренный нами способ конечного порождения множества T - представление вычислимой функции конечным операторным термом. Существуют и другие способы

представления вычислимой функции. Интерес к представлению алгоритма множеством функциональных термов определяется тем, что в этом представлении, в отличие от программы, не фиксируется способ/порядок исполнения операторов языка программирования, не задается распределение ресурсов, что оставляет много свободы для эффективной параллельной реализации алгоритма.

2.ЗАДАЧА КОНСТРУИРОВАНИЯ ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ

2.1.Представление алгоритма

Для точной формулировки задачи прежде всего определим понятие представления алгоритма *A*, вычисляющего функцию F_A [5]. Рассматриваться будут лишь такие представления алгоритма, в которых в явном виде используются преобразователи значений входных величин (переменных) в значения выходных. Такой преобразователь изображает элементарный шаг в интуитивном понятии алгоритма [10,13]. Представление алгоритма, которое может быть исполнено компьютером, называется *программой*. Обычно программа содержит назначение ресурсов для реализации всех объектов алгоритма.

Каждый преобразователь *a* (будем впредь называть его *операцией*) вычисляет функцию f_a, значение которой есть результат выполнения преобразования (выполнение операции) *a*.

Представление алгоритма *A* есть набор *S=(X,F,C,M),* где *X={x,y,...,z}* - конечное множество переменных, *F={a,b,...,c}* - конечное множество операций.

С каждой операцией $a \in F$ связаны входной *in(a)* и выходной *out(a)* наборы переменных из *X*. Наборы *in(a)* и *out(a)* будут при необходимости рассматриваться и как множества со всеми определенными для множеств операциями. Две операции *a* и *b* называются *информационно зависимыми*, если $out(a) \cap in(b)$

непусто. Две операции a_1 и a_n называются *транзитивно информационно зависимыми*, если существует последовательность a_1, a_2,…, a_n операций, попарно информационно зависимых.

Графическое представление операции a показано на рис. 2.1. Мы будем говорить, что операция a вычисляет переменные $out(a)=(y_1,...,y_m)$ из переменных $in(a)=(x_1,...,x_n)$.

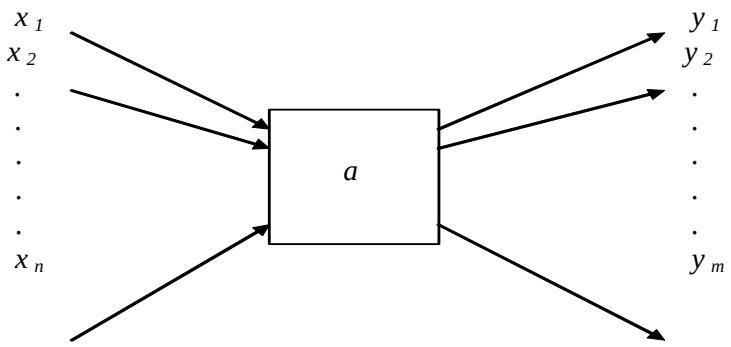


Рис. 2.1.

Отметим здесь, что функция f_a имеет n аргументов и m выходных переменных в отличие от обычно рассматриваемых в теории алгоритмов функций с одной выходной переменной. Такой вид операций более подходит для рассмотрения в программировании, поскольку далее мы договоримся о том, что операция в программе реализуется процедурой и, следовательно,

необходимо изображать преобразователи со многими выходными переменными.

C - *управление*, т. е. множество ограничений на порядок выполнения операций. Управление может задаваться разными способами. Здесь полагаем, что каждое ограничение определяет порядок исполнения пары операций вида $(a<b)$, что означает, что операция a должна стартовать и завершиться раньше b. Таким образом, управление C – это бинарное отношение частичного порядка на множестве F операций алгоритма A.

M - функция, задающая отображение множеств переменных и операций в физические устройства параллельной вычислительной системы (здесь обычно мультикомпьютер), т. е. *M* задает *распределение ресурсов* мультикомпьютера.

Более детальные определения *C* и *M* не приводятся, а их свойства и формы представления лишь поясняются на примерах.

Для определенности считается, что для выполнения операции a в программе используется одноименная процедура a, у которой есть входной $in(a)=(x_1,...,x_n)$ и есть выходной $out(a)=(y_1,...,y_m)$ наборы переменных, в программе обращение к a имеет вид $a(x_1,...,x_n,y_1,...,y_m)$.

Отображение *M* может сопоставить переменным x_1 и y_1 одну и ту же ячейку памяти, а переменным x_2, ..., x_n, y_2, ..., y_m сопоставляются, например, отдельные ячейки памяти каждой. Обращение тогда может иметь вид $a(x_1, ..., x_n, y_2, ..., y_m)$, имея в

виду, что переменная программы x_1 попеременно содержит значения переменных алгоритма x_1 и y_1, как это обычно и делается в последовательных программах.

Одно из ограничений C может, к примеру, требовать, чтобы операция b начала выполняться после завершения операции a. Часть ограничений управления C, определенных *информационной зависимостью* между операциями (см. пример **П2**), называется *потоковым* управлением остальные ограничения C задают *прямое* управление, связанное обычно с распределением ресурсов и/или оптимизацией исполнения программы.

Отображение M назовем *тождественным*, если оно ставит в соответствие каждой переменной алгоритма отдельную ячейку памяти, а каждой операции - собственный, ни с кем не разделяемый процессор.

Реализацией алгоритма A, представленного в форме S, называется выполнение операций в некотором произвольном допустимом порядке, который не противоречит управлению C. Предполагается, что при любой реализации вычисляется функция F_A, т.е. C всегда содержит потоковое управление, которое и гарантирует вычисление F_A.

Множество всех реализаций алгоритма A, представленного в форме S, обозначим $P(A,S)$. Если $P(A,S)$ есть одноэлементное множество, то S - *последовательное*

представление. Будем говорить, что *S* - *параллельное* представление алгоритма *A*, если множество *P(A,S)* содержит, более одной реализации.

П1. Алгоритм задан на языке С, каждый вычисляющий (не управляющий) оператор программы есть операция, выполнение операции есть выполнение оператора языка, порядок выполнения операций полностью фиксирован и, следовательно, представление последовательное.

П2. Пусть $F=f(x_1,x_2)$, $x_1=g(y_1,...,y_k)$, $x_2=h(z_1,...,z_m)$ - операции, $out(h)=\{x_2\}$, $out(g)=x_1$, $out(f)=\{f0\}$. Тогда алгоритм *A* вычисления функции $F=\mathbf{S}(f,g,h)= f(g_1(y_1,..., y_k), h=(z_1,...,z_m)$ представлен в параллельной форме S_1, *f*, *g* и *h* – операции (рис.2.2а).

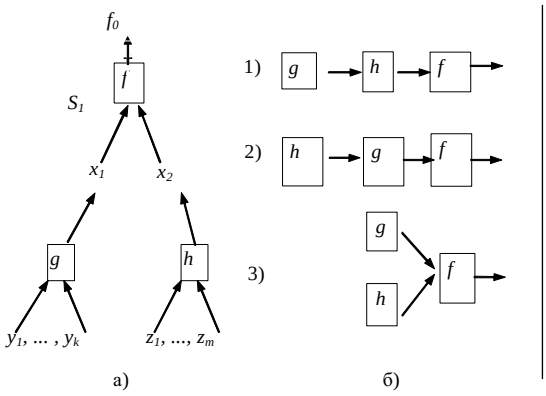


Рис. 2.2.

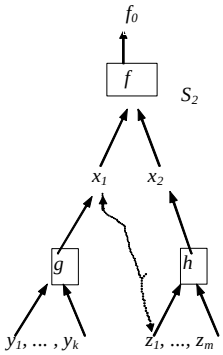


Рис. 2.3.

На рис. 2.2а стрелки показывают информационные зависимости между операциями. Следовательно, управление C в представлении S_1 - это множество $\{(g<f), (h<f)\}$.

Допустим всякий порядок выполнения операций, при котором вычисление аргументов операции предшествует выполнению операции, управление C - потоковое, сохраняющее необходимые информационные зависимости между операциями.

Уменьшить потоковое управление (например, разрешить операции f выполняться раньше операции g) нельзя, так как при этом будет вычисляться, вообще говоря, не функция F, а какая-то другая. В этом смысле потоковое управление есть минимальное управление, гарантирующее вычислений функции F. Множество реализаций алгоритма A, представленного в форме S_1, состоит из трёх элементов, показанных на (рис.2.2б).

Определим теперь бинарное отношение непроцедурности на множестве представлений алгоритма A. Будем говорить, что представление S_1 алгоритма A более *непроцедурно*, чем S_2, если $P(A,S_1) \supset P(A,S_2)$. Например, пусть для хранения значений переменных используются ячейки памяти и заданы представления $S_1=(X,F,C_1,R_1)$ и $S_2=(X,F,C_2,R_2)$ алгоритма вычисления функции $f=f(g(y_1,...,y_k),h(z_1,...,z_m))$ (рис. 2.3). В S_2 потребуем, чтобы значения переменных алгоритма x_1 и z_1 хранились в одной и той же ячейке памяти.

Тогда $P(A,S_1) \supset P(A,S_2)$, так как в силу сделанного распределения памяти для вычисления f прямое управление в C_2 должно теперь содержать требование, чтобы операция h выполнялась раньше g, то есть управление C_2 в представлении S_2 - это множество $\{(g<f), (h<f), (h<g)\}$. При этом операция h, начав исполняться, использует значение переменной z_1 до того, как в ту же ячейку памяти будет записано значение переменной x_1, вычисленное операцией g.

Очевидно, что представление S алгоритма A с потоковым управлением C и тождественным распределением ресурсов R является *максимально непроцедурным* представлением A. Такое представление будем называть просто *непроцедурным*.

Рассмотрим еще несколько примеров программ, иллюстрирующих введенные определения.

П3. Алгоритм A задан в форме программы P, записанной на некотором последовательном языке программирования. Операторы языка, вычисляющие значения переменных, есть операции. Прямое управление задается последовательностью операторов в программе либо специальными операторами управления типа **go to.**

Порядок выполнения операций полностью фиксирован. Распределение памяти задаётся идентификаторами переменных программы, которые фактически есть имена ячеек памяти, имена переменных алгоритма в программе не сохраняются.

Предполагается, что для выполнения алгоритма может быть использован только один процессор.

П4. На рис. 2.4. приведен пример представления *S=(X,F,C,M)* алгоритма *A* и одного из его возможных представлений в форме программы *P*. Здесь *X={x,y,z,x1,z1}, F={a,b,c}*. *C* - потоковое управление (показано стрелками), *M* - тождественное распределение ресурсов.

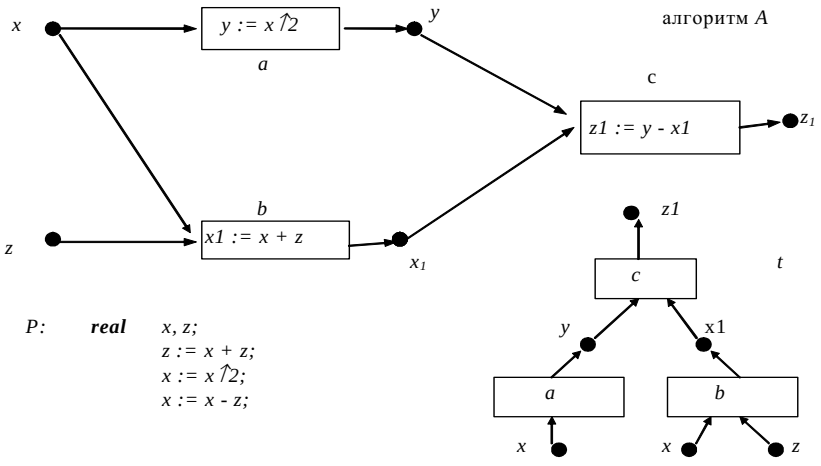


Рис. 2.4.

Программа *P* может быть построена следующим регулярным способом. Вначале строится функциональный терм *t* (рис.2.4.), определяющий алгоритм решения задачи. Устройство терма показывает, что после выполнения операции *b* значение переменной *z* более не используется, а значение переменной *x* еще будет нужно для корректного исполнения операции *a*.

Поэтому переменной *x1* может быть назначена та же ячейка памяти, что и для переменной *z* (что и сделано в программе *P*).

После исполнения операции *a* значение переменной *x* тоже становится не нужным и потому для хранения значения переменной *y* может быть назначена та же самая ячейка, что и для переменной *x*.

После исполнения операции *c* сразу две переменные - *y* и *x1* - становятся не нужными (их значения полностью использованы) и потому значение переменной *z1* может быть положено как в ячейку, где хранится значение *y*, так и в ячейку, где хранится значение переменной *x1* (в программе *P* значение переменной *z1* положено в ту же ячейку, где хранилось значение переменной *y*, а еще ранее хранилось значение переменной *x*).

Таким образом, в программе *P* переменная программы, помеченная идентификатором *x* (имя ячейки памяти), последовательно хранит значения переменных алгоритма *x*, *y*, z_1 (т.е. задано отображение $M(x)= M(y)=M(z1)=x$, последний *x* есть имя ячейки), порядок выполнения операторов программы не может быть изменен в силу выбранного распределения памяти.

При необходимости реализовать алгоритм *A* на мультикомпьютере программа *P* распараллеливается, т.е. преобразуется в функционально эквивалентную программу P_1 (вычисляющую ту же функцию F_A), но допускающую параллельное выполнение некоторых операторов. Таким образом,

задача распараллеливания программы заключается в нахождении параллельного представления *A* (конструировании новых *C* и *M*) на основе анализа *P*.

П5. Алгоритм *A* задан в форме программы *P*, записанной на некотором параллельном языке, использующем конструкции типа **fork** и **join** для задания параллельных ветвей.

Например **fork** V_1, V_2, V_3 **join**, где V_1 , V_2 , V_3 - последовательные участки программы, которые могут выполняться независимо. Участки V_1 , V_2 , V_3 формируются таким образом, чтобы объём вычислений в них (а значит и время их выполнения) был одинаковым. Схематично исполнение программы показано на рис. 2.5, стрелки показывают поток управления. После оператора **fork** ветви V_1, V_2, и V_3 могут исполняться в произвольном порядке, независимо друг от друга. Оператор **join** ожидает завершения всех ветвей.

Программа хорошо выполняется на трех одинаковых процессорах и вдвое хуже на двух в силу излишне жестко заданного прямого управления. Во втором случае вначале будут выполнены две (доступны два процессора) любые ветви, например V_1 и V_3, затем ветвь V_2 (один процессор простаивает) и лишь после этого оператор **join** разрешит продолжить исполнение программы *P* далее. Программа *P* должна преобразовываться при изменении числа доступных процессоров

мультикомпьютера для достижения хорошего качества исполнения.

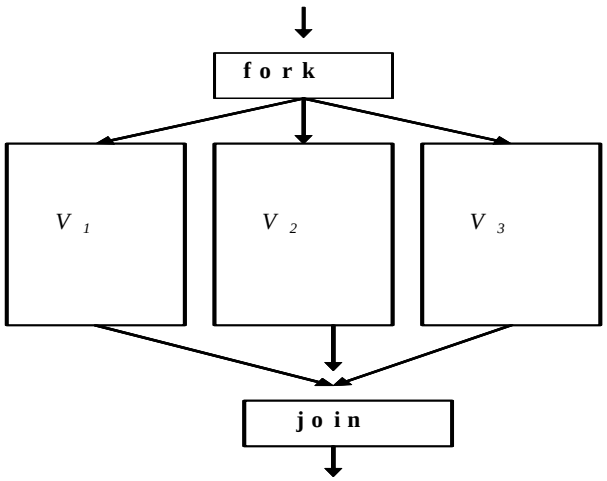


Рис. 2.5.

П6. Алгоритм *A* задан асинхронной программой *P* (см. параграф 4.1). Управление *C* задается спусковыми функциями и/или *сетями Петри* (см. параграф 3.3). Распределение ресурсов находит отражение в устройстве *C*, поэтому, хотя *P* и в меньшей степени зависит от изменений в структуре мультикомпьютера в силу недетерминированности управления, ее также часто необходимо преобразовывать (конструировать новые *C* и *M*) при изменении структуры мультикомпьютера для получения хороших рабочих характеристик реализации *A*.

Обращаем внимание читателя на диаграмму (рис. 2.6). Она показывает, что для каждой вычислимой функции F_A существует счетное множество различных алгоритмов $A_1, \ldots A_n, \ldots$, вычисляющих F_A, и для каждого алгоритма A_n существует счетное множество различных программ $P_1, \ldots, P_m, \ldots$, реализующих A_n.

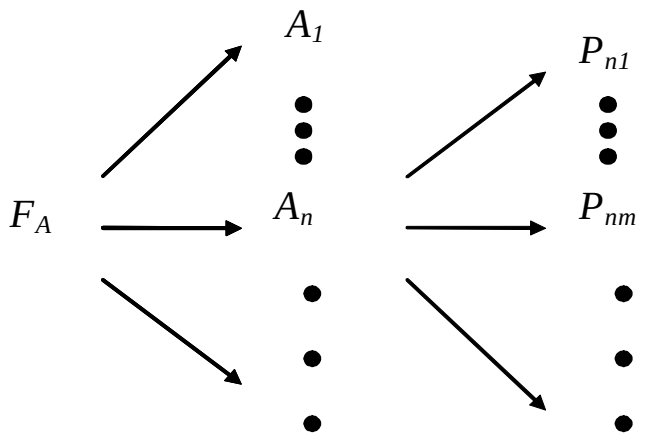


Рис. 2.6.

Эта диаграмма проясняет вопрос о том, какие операции реализуемы в оборудовании и/или в языках программирования. Например, во всех процедурных языках программирования и в оборудовании есть хорошо реализуемые операции умножения целых и вещественных чисел, но нет операции умножения матриц. Дело в том, что в ограничениях оборудования для реализации функций умножения любых представимых в оборудовании целых и вещественных чисел существуют

субоптимальные алгоритмы, которые незначительно отличаются друг от друга по сложности. Поэтому эти алгоритмы можно встраивать в оборудование и/или в компилятор. Алгоритмы же умножения матриц имеют существенно разное качество реализации в зависимости от свойств перемножаемых матриц, почему эти алгоритмы и невозможно встраивать ни в оборудование, ни в языки программирования без неприемлемо больших потерь в производительности программы.

Эта же диаграмма вносит разъяснения в вопрос о том, чем опытный квалифицированный программист отличается от начинающего. Опытный программист на основании своего опыта в состоянии сконструировать (скомпоновать) хороший алгоритм решения задачи с использованием широкого множества известных ему алгоритмов, тогда как неопытный программист делает выбор из того сравнительно небольшого числа алгоритмов, что он успел узнать. Так как этот выбор у начинающего невелик, то и качество программы получается похуже, и программирование нередко кажется начинающему простым делом.

Ситуацию образно демонстрирует процесс создания романа Л.Толстого «Война и мир». Любой русскоязычный читатель знает все слова в романе. Теоретически он мог бы взять эти слова и без проблем составить из них текст знаменитого романа. Однако только Льву Толстому это удалось.

Конструирование алгоритмов и программирование, в этом смысле, мало отличаются от писательства. Здесь тоже есть простейшие вычислимые функции и три оператора для конструирования новых алгоритмов. Надо только применить их в должном порядке, и задача решена.

В приведенном определении представления алгоритма использовано лишь конечное число операций, при этом в реализующей программе отсутствуют циклы. Для обсуждения и определения новых введенных понятий это было не существенно. Использование *массовых* операций легко исправляет этот недостаток.

Не вдаваясь в детали определения массовых операций (более детально см. в [5]), рассмотрим алгоритм A_{file} ввода записей файла f в память (в массив y), затем обработку всех его компонент операцией a, посылку результата в компоненты массива z. С учетом того, что в максимально непроцедурном представлении должен быть указан каждый вычислительный акт, каждое возможное выполнение операции, алгоритм будет иметь вид, показанный на рис. 2.7. Здесь операция *read(f, i)* считывает i-ю запись файла в i-й компонент массива $y[i]$, операция a^i обрабатывает i-ю запись массива $y[i]$, а результат записывает в i-ю запись файла z.

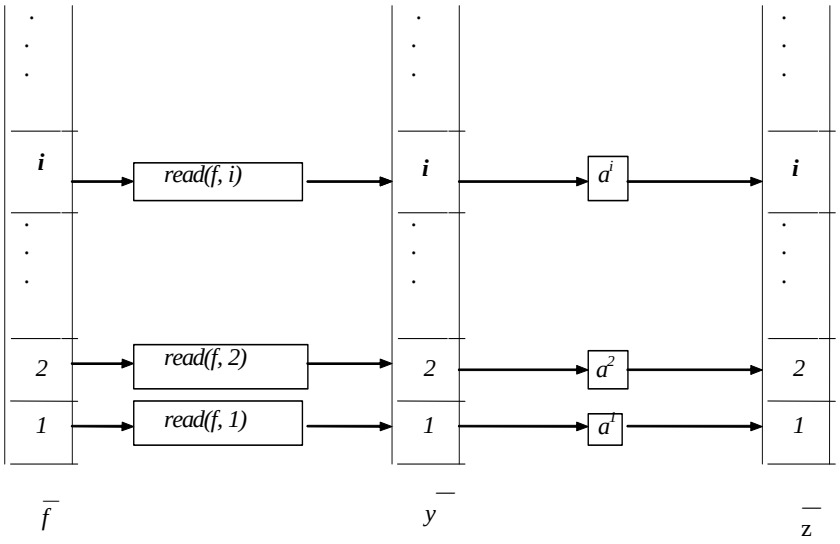


Рис. 2.7.

Если отображение *M* сопоставляет массиву *y* участок памяти, в котором может быть размещена лишь одна запись, и доступен только один процессор, тогда прямым управлением *C* должен быть задан следующий жестко определенный порядок выполнения операций: *read(f,1)*, a^1, *read(f,2)*, a^2, ... (обычно в программе задается с использованием оператора цикла).

При таком порядке исполнения считанная запись файла будет потреблена операцией *a*, память освобождена, и следующая запись может быть считана в тот же участок памяти, что и предшествующая запись. Алгоритм A_{file} будет правильно реализован, т.е. будет вычислена функция F_{Afile}.

Если при тех же условиях на размещение массива *y* для исполнения алгоритма доступны два процессора, тогда, кроме последовательного, может быть ещё организовано конвейерное исполнение алгоритма (другая его реализация), при котором обработка *i*-ой записи файла делается одновременно со считыванием (*i+1*)-ой записи файла (рис. 2.8).

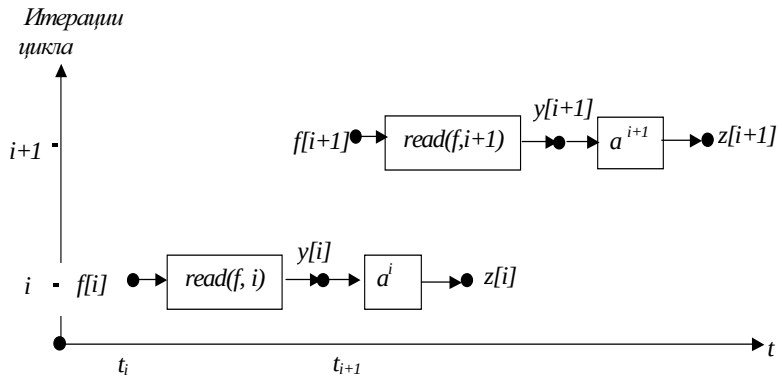


Рис. 2.8.

Если разрешается разместить в памяти две записи, тогда может быть использован режим ввода с двумя буферами и две операции – a^{2i-1} и a^{2i} - смогут выполняться одновременно, когда есть доступные процессоры и т.д.

Языки программирования, которые содержат лишь средства задания алгоритма, т.е. множества функциональных термов, можно назвать *функциональными языками*. В них должны быть средства для задания множества операций и

переменных алгоритма, а также потоковое управление. Это просто языки выражений.

Языки программирования единственного присваивания (single assignment) не имеют таких средств распределения памяти, которые бы позволили присвоить разные значения одной переменной. В них нет средств для задания вычислений типа $x:=x+1$ (напомним, что этот оператор присваивания размещает значения двух разных переменных алгоритма в одной переменной программы). Это присваивание записывается только в виде $x_{i+1}:=x_i+1$. Программа в языках единственного присваивания, не обремененная информацией о распределении ресурсов, гораздо легче поддается формальному анализу.

Но конечно, это создает большие проблемы с реализацией этих языков, так как на конечных ресурсах необходимо в ходе вычислений разместить значения в общем случае потенциально бесконечного множества переменных алгоритма. В компиляторе/интерпретаторе необходимо применять некоторую стратегию использования участков памяти, ранее занятых одними переменными, которые уже выполнили свою роль и более не будут востребованы в программе, для размещения значений других переменных, т.е. выполнять распределение ресурсов. Важная стратегия состоит в нахождении такого порядка выполнения операций, при котором выработанные значения переменных как можно быстрее потребляются другими

операциями и далее не требуется, допуская повторное использование ячеек памяти. При ручном программировании программист обычно делает это довольно эффективно и часто почти не задумываясь.

Языки, в которых дополнительно к функциональным средствам есть ещё и средства для задания прямого управления и распределения ресурсов, называются *процедурными* языками. Существует целый спектр языков разной степени непроцедурности от императивных языков типа C с жестким прямым управлением, допускающем единственную реализацию, до максимально непроцедурных (собственно уже функциональных) языков.

2.2.Требования к представлению параллельного алгоритма.

Примеры показывают, что к представлению *S* алгоритма *A* для последующей реализации предъявляются противоречивые требования.

С одной стороны, необходима непроцедурность *S*, так как она уменьшает зависимость *S* от свойств конкретного мультикомпьютера, позволяет качественно реализовать алгоритм на различных мультикомпьютерах, а значит, есть возможность накапливать программный фонд для них.

С другой стороны, есть потребность в таком уменьшении непроцедурности *S*, при котором в множестве *P(A,S)* остаются лишь оптимальные относительно некоторого критерия

реализации, так что уменьшение непроцедурности S будет способствовать улучшению рабочих характеристик программы (уменьшатся затраты на реализацию управления, которые имеют тенденцию стремиться к 100%).

Противоречие может быть разрешено, если конструирование параллельной программы решения задачи проводить в два этапа:

1).Вывод (конструирование) алгоритма решения задачи и его непроцедурное представление. Таким базисным представлением алгоритма в параллельном программировании является множество функциональных термов.

2).Конструирование по непроцедурному представлению алгоритма и описанию мультикомпьютера прямого управления и распределения ресурсов такой степени непроцедурности, которая позволит эффективно реализовать алгоритм на этом именно мультикомпьютере. Конструирование заключается в «свертке» множества функциональных термов в желаемую конечную программу, с должными управлением и распределением ресурсов.

При этом в программном фонде хранятся алгоритмы, представленные в непроцедурной форме, которая перед выполнением алгоритма на конкретном мультикомпьютере с конкретными входными данными дополняется соответствующим прямым управлением и не тождественным распределением

ресурсов для получения оптимальных относительно заданного критерия рабочих программ.

В такой общей постановке задача конструирования эффективной параллельной программы (конструирования *C* и *M*) в целом не имеет приемлемого решения, хотя к настоящему времени разработано много алгоритмов конструирования *M* (*M*-алгоритмов). Слишком велико оказалось разнообразие различных типов и структур алгоритмов и оборудования.

Какую систему параллельного программирования (СПП) хотелось быть иметь для мультикомпьютеров? Прежде всего, СПП должна быть достаточно проста, так чтобы любой прикладной программист, смог сам без чрезмерных усилий создавать эффективные параллельные программы. Для этого СПП должна быть основана на некотором широко распространенном процедурном языке программирования, например С, в который необходимо добавить непроцедурные средства представления *A* таким образом, чтобы усилия пользователя и СПП взаимно дополняли бы друг друга в попытках организовать эффективные параллельные вычислений.

Термин ‘’эффективный’’ используется, конечно, неформально. Он может пониматься, например, так, что выгоды от использования мультикомпьютера компенсируют прикладному пользователю затраты на разработку параллельной программы, а потери производительности мультикомпьютера от

использования СПП высокого уровня оказываются приемлемыми за счет повышения производительности труда программистов.

Задача отображения алгоритма *A* на ресурсы мультикомпьютера может решаться статически, до начала вычислений. В этом случае все принятые решения по распределению вычислительных ресурсов, в первую очередь процессоров, фиксируются в тексте программы.

Такой подход, однако, неприемлем во многих случаях. Так, мультикомпьютер может иметь много ресурсов и далеко не каждая программа все их использует. Следовательно, вычислительные ресурсы мультикомпьютера надо разделять динамически, т.е. реализовать мультипрограммирование на вычислительных ресурсах мультикомпьютера.

В этом случае, *M*-алгоритм должен состоять их двух частей: статической, которая реализуется в трансляторе и производит распределение той части ресурсов мультикомпьютера, которая может быть отведена программе до начала вычислений (часть оперативной памяти, регистры и т.п.), и динамической, которая реализуется в программе или в операционной системе и производит захват и перераспределение ресурсов мультикомпьютера в ходе исполнения прикладной программы (процессоры, каналы, оперативная и вторичная память, регистры).

2.3. Простейшая программа, реализующая алгоритм

Рассмотрим подробнее проблему поиска подходящего представления алгоритма A на примере построение простой параллельной программы, реализующей A.

Пусть алгоритм A задан конечным множеством термов $\{t_1, t_2,\ldots, t_n\}$, в термах используются переменные $\{x_1, x_2, \ldots, x_m\}$ и операции $\{a_1, a_2, \ldots, a_k\}$. Значения всех входных переменных известны, значения остальных вырабатываются операциями алгоритма. Все переменные x_i, $i=1,2,\ldots,m$, объявляются глобальными. Обозначим T_i имя процедуры, реализующей терм t_i. Тогда искомая программа выглядит так (процедуры T_i могут выполняться в любом порядке):

var $x_1, x_2, \ldots, x_m$;
var $y_1, y_2, \ldots, y_n := 0$;
var $z_1, z_2, \ldots, z_k := 0$;
fork
if $y_1 = 0$ **then** (**call** T_1; $y_1 := 1$);
if $y_2 = 0$ **then** (**call** T_2; $y_2 := 1$);
. . . . . .
if $y_n = 0$ **then** (**call** T_n; $y_n := 1$);
join
end

Здесь $y_1, y_2, \ldots, y_n$ - управляющие переменные, которые инициируются значением 0. Каждой процедуре T_j соответствует управляющая переменная y_j, $j=1,2, \ldots, n$.

Каждой процедуре a_i сопоставляется управляющая переменная z_i, $i=1,2, \ldots, k$, нулевое значение которой означает, что операция a_i еще не выполнялась. Предикат $P_{in}(ak)$ принимает значение *True* (истина), если значения всех входных переменных операции *ak* заданы, и *False* (ложь) в противном случае. Может быть сконструирована следующая процедура T_j:

$\forall s=1, \ldots, r_j$ (**if** ($P_{in}(as)\&z_s=0$) **then** (**call** *as*; $z_s:=1$)),

где $as, s=1, \ldots, r_j$, - операции, используемые в терме T_j.

Конечно, эта программа чаще всего окажется нехорошей. При ее конструировании не учитывались никакие технологические требования, такие, например, как надежность исполнения, минимизация потребляемых ресурсов, удобный интерфейс и многие другие.

2.4.Сравнительная непроцедурность языков программирования.

Рассмотрим понятие непроцедурности языков программирования, чтобы иметь возможность сравнивать их по этому свойству.

Прежде всего отметим, что одной и той же конструкции языка, определенного некоторой грамматикой, можно сопоставлять разные правила реализации, т.е. задавать для нее разные операционные семантики.

Наоборот, для разных конструкций языка могут быть определены правила исполнения, порождающие одно и то же множество реализаций.

И в третьих, при сравнении языков с разными правилами порождения языковых конструкций и с разными правилами их реализации следует исходить из некоторого соответствия между конструкциями языка. Уточним теперь эти три концепции.

Пусть L - язык программирования, для которого заданы две операционные семантики S_1 и S_2, определенные как множества реализаций $S_i(k)$, i=1,2, где $k \in L$ - конструкция языка L. Операционная семантика определяет, собственно, возможные реализации конструкций языка. Именно операционная семантика выражает процедурность языка L, в отличие от семантики математической, связанной с непроцедурностью. Будем считать, что семантика S_1 более непроцедурна, чем S_2, если $\forall k \in L$ ($S_1(k) \supset S_2(k)$), что соответствует нашему определению понятия непроцедурности представления алгоритма.

Если L - язык программирования, k_1 и k_2 - его конструкции, S - операционная семантика, то скажем, что k_1 более непроцедурна, чем k_2 в смысле семантики S, если $S(k_1) \supset S(k_2)$.

П1.Пусть L - язык программирования, например, Фортран или С, S_1 - его семантика. Определим теперь его семантику S_2, совпадающую с S_1 всюду, кроме операторов цикла, для которых допустим в S_2 параллельное выполнение информационно

независимых *итераций*[1] цикла. Тогда семантика S_2 более непроцедурна, чем S_1 . В частности, цикл

x[1]:=c1; x[2]:=c2;
for *i* =1 **to** 100 **do**
(*x[i+2]:=a(x[i]); x[i+3]:=b(x[i+1];*)

определяет выполнение операций *a* и *b* в следующем порядке: *a(x[1]), b(x[2], a(x[3]), b(x[4]...* . Так как в семантике S_2 допускается параллельное выполнение независимых итераций цикла, то допустим следующий (наряду со счетным множеством других) порядок выполнения итераций цикла:

1: *a(x[1]), b(x[2]*
2: *a(x[3]), b(x[4]*
3: . . .

Если дополнить язык С циклом вида $\forall x \in X | P(x)$::A*;* где *X* - массив, *A* – тело цикла, операционная семантика оператора цикла не фиксирует порядок выполнения итераций и допускает выполнение итерации, если вычислены все ее аргументы. Тогда мы скажем, что введенный оператор цикла более непроцедурен, чем цикл типа **for**.

Пусть теперь L_1 и L_2 - языки программирования с семантиками S_1 и S_2 соответственно, *f* - отношение

[1] Итерацией цикла называется исполнение тела цикла с некоторым конкретным значением параметра цикла

функциональной эквивалентности, заданное на $K(L_1)\times K(L_2)$, где $K(L)$ - множество конструкций (операторов) языка программирования L. Дадим определения сравнительной непроцедурности языков L_1 и L_2.

Определение 1. Язык L_1 более непроцедурен, чем L_2, если

$$\overset{f}{\forall}\, k_2 \in L_2 \; \exists k_1 \in L_1 \; ((k_1 \sim k_2) \& (S_1(k_1) \supset S_2(k_2)))$$

В соответствии с определением **1** язык L_1 считается более непроцедурным, чем L_2, если любая программа на языке L_2 может быть записана на языке L_1 в более непроцедурной форме.

Определение 2. Язык L_1 более непроцедурен, чем L_2, если

$$\overset{f}{\forall}\, k_2 \in L_2 \; \forall k_1 \in L_1 \; ((k_1 \sim k_2) \rightarrow (S_1(k_1) \supset S_2(k_2)))$$

Согласно определению 2 язык L_1 более непроцедурен, чем L_2 в случае, если любая программа на L_2 неизбежно записывается на L_1 (алгоритм представляется в L_1) в более непроцедурной форме.

Определение 3. Пусть задано некоторое отображение конструкций $h : K(L_1) \rightarrow K(L_2)$. Если

$$\overset{f}{\forall}\, k_2 \in L_2 \; \exists k_1 = h^{-1}(k_2) \; ((k_1 \sim k_2) \& (S_1(k_1) \supset S_2(k_2)))$$

то язык L_1 более непроцедурен, чем L_2 в смысле h.

III. ВЗАИМОДЕЙСТВУЮЩИЕ ПРОЦЕССЫ

Программа *P* для мультикомпьютера описывает систему взаимодействующих процессов и именно эта система процессов рассматривается в настоящей главе. Понятно, что вначале программа *P* загружается для исполнения в процессорные элементы (ПЭ) мультикомпьютера. При исполнении программа *P* порождает описанные в ней процессы, они в свою очередь назначаются на исполнение на процессорные элементы мультикомпьютера и тоже могут порождать процессы. Процессы должны иметь возможность обмениваться данными и синхронизовать свое исполнение. Для уточнения всех этих понятий рассмотрим вначале общую модель параллельной программы для мультикомпьютера.

3.1.Последовательные процессы.

Программа *P* - объект статический и она именно описывает множество процессов и их взаимодействия. Процесс - объект динамический, он возникает только при исполнении программы. *Процессом (последовательным процессом)* называется исполняющаяся программа (*программа процесса*) со своими входными данными.

Первоначально программа *P* запускается на счет как один процесс, который обычно исполняется на управляющем компьютере (host computer) мультикомпьютера. Программа *P* фрагментирована, в ней выделяются фрагменты кода, способные

выполняться независимо. При исполнении фрагменты *P* запускаются на счет как отдельные процессы (процессы *порождаются*) со своими входными данными. Для этого программный код и необходимые данные процесса пересылаются в выделенный для исполнения процесса процессорный элемент (ПЭ), процесс требует и получает необходимые для своего исполнения ресурсы и наконец стартует. Фрагменты *P* составляют программы процессов.

Порождение процесса делается специальным оператором языка программирования, исполнение которого может реализоваться системным вызовом, т.е. обращением к операционной системе (ОС) за выполнением этой работы. Программа процесса - это всем известная последовательная программа, со всеми необходимым для счета переменными и управлением. Переменные процесса, как правило, недоступны другим процессам. Все ее операторы исполняются последовательно в порядке, определенном языком программирования. Программа процесса сейчас, как правило, разрабатывается на языке С.

Одна и та же исполняющаяся программа с разными входными данными определяет разные процессы. Разные исполняющиеся программы также определяют разные процессы. Каждый процесс в ходе исполнения может быть полноправным клиентом ОС, потреблять для своего исполнения ресурсы (ПЭ,

память, устройства ввода/вывода и т.п.), обмениваться информацией с другими процессами, конкурировать за ресурсы, порождать процессы, завершаться.

Ясно, что в каждом ПЭ должна размещаться некоторая более или менее развитая ОС (это обычно одна из версий UNIX) для организации исполнения в ПЭ одного или более процессов. В совокупности эти ОС процессорных элементов составляют распределенную ОС мультикомпьютера. Для управления процессами ОС создает для каждого процесса *блок управления процессом*, который содержит описание текущего состояния процесса (имя, ресурсы, отношение к другим процессам и т.д.).

Процессы могут порождаться и внутри программы, без участия ОС, так как порождение процесса в ОС - довольно времяемкая процедура, что не позволяет создавать небольшие процессы.

Состояние процесса в общем случае определяется значением всех переменных, определенных или связанных с процессом, в частности, значениями переменных, определенных ОС для организации выполнения процесса (например, переменные блока управления процессом). Далее в этой главе будут рассматриваться только взаимодействия между процессами, поэтому понятие состояния процесса для простоты ограничивается и определяется как значение особой переменной

состояние, принимающей одно из следующих допустимых значений:

- **`исполняется`** - команды программы процесса исполняются, процесс активен,
- **`ожидает`** - процесс ждет совершения некоторого события,
- **`готов`** - процесс имеет все необходимые ресурсы и готов продолжить исполнение.

В ходе исполнения параллельной программы процессы могут порождаться и завершаться (для этого в языках параллельного программирования есть специальные операторы). Процесс, породивший новый процесс, называется *родительским* процессом по отношению к порожденному новому процессу. Порожденные процессы называется *детьми* родительского процесса.

Порожденные процессы-дети в свою очередь могут порождать новые процессы, по отношению к которым они выступают в качестве родительских процессов и таким образом программа P распространяется по ПЭ мультикомпьютера. Доступные ПЭ выделяются программе P в момент начала вычислений либо несколько программ динамически конкурируют за ПЭ. Каждый ПЭ отдается в монопольное использование одной из программ.

Допускаются разные способы организации выполнения родительских процессов и их детей.

1. Порядок выполнения процессов.

Родительский процесс порождает новые процессы и:

- продолжает свое исполнение параллельно с процессами-детьми,
- задерживает свое исполнение и ожидает того момента, когда завершатся все порожденные им процессы-дети.

2. Использование ресурсов.

Родительский процесс и процессы-дети:

- разделяют общие ресурсы,
- все процессы имеют свои собственные не разделяемые ресурсы.

3. Завершение.

- Своей командой родительский процесс может завершить выполнение своих процессов-детей. Для такого завершения может быть несколько причин: работа процессов-детей более не нужна, либо они потребляют слишком много ресурсов и не дают работать другим процессам, либо в них случились события, не позволяющие продолжить вычисления (например, нет затребованного ресурса).
- Процессы-дети не могут завершить свой родительский процесс, хотя, конечно, могут служить причиной, по которой родительский процесс примет решение о своем завершении.

- Для принятия решения о завершении процессов-детей родительский процесс должен знать *состояние* своих детей, таким образом, состояние процессов-детей доступно родительскому процессу.
- Если родительский процесс завершается, то как правило, завершаются и все его дети. ОС должна отследить эту ситуацию, что удобно при программировании и отладке (не надо беспокоиться о «зависших» процессах).

В зависимости от возможностей влиять на ход исполнения других процессов все множество процессов разбивается на *независимые* и *взаимодействующие* процессы [9]. Полномочия процессов влиять друг на друга определяются при их порождении.

Независимые процессы не могут влиять на выполнение других процессов (передавать данные, изменять значения переменных, останавливать и т.п.) и их исполнение не может управляться из других процессов. А потому:

- их состояние недоступно другим процессам,
- их исполнение и результат зависят только от входных данных (отсутствует характерный для исполнения параллельных программ недетерминизм),
- их исполнение может быть задержано ОС без оказания влияния на работу остальных процессов.
- они не разделяют ресурсы и данные с другими процессами.

Таким образом, независимые процессы полностью автономны. В частности, процессы, порожденные разными программами P_1 и P_2, выполняются независимо друг от друга.

Процессы, которые совместно выполняют общую работу, должны и могут влиять друг на друга. Такие процессы:

- имеют взаимный доступ к переменной *состояние*.
- их исполнение недетерминировано. При каждом новом исполнении такие процессы могут по разному влиять друг на друга (хотя бы только по времени) и, следовательно, вырабатывать в случае ошибки разные результаты.
- их исполнение в общем случае невозможно повторить. Каждое исполнение системы процессов происходит по-разному, что крайне затрудняет отладку.

Параллельно протекающие процессы программы *P* могут выполняться в одном и том же ПЭ, разделяя ПЭ в режиме мультипрограммирования, либо на разных ПЭ мультикомпьютера, общаясь друг с другом посредством обмена данными.

Процессы взаимодействуют, передавая друг другу данные и синхронизируя ход вычислений. Здесь предполагается, что передача данных между процессами осуществляется через особые *разделяемые* переменные, доступные всем или подмножеству процессов программы *P*. Одни процессы присваивают значение разделяемым переменным, а другие в

нужный момент (после некоторого ожидания, если потребуется) могут считать эти значения.

Разделяемые переменные реализуются по-разному для разным мультикомпьютеров. В мультикомпьютерах с разделяемой памятью (*shared memory*), которые часто называются также *мультипроцессорами* (рис. 3.1.), разделяемые переменные могут быть реализованы как участки памяти, доступные нескольким процессам, где они могут оставлять и забирать значения переменных. Такая реализация легко осуществима, так как всем процессорам доступна вся память мультикомпьютера.

В мультикомпьютерах с распределенной памятью (*distributed memory*) разделяемые переменные обычно реализуются в виде логического канала для передачи данных между процессами. Здесь ПЭ соединены коммутационной сетью, все взаимодействия между процессами сильно зависят и по-разному реализуются в зависимости от ее структуры. Большое разнообразие структур коммутационной сети значительно затрудняет программирование мультикомпьютеров с распределенной памятью. На рис. 3.2 показан мультикомпьютер со структурой решетка (более детально с архитектурами современных вычислителей можно ознакомиться в главе 9).

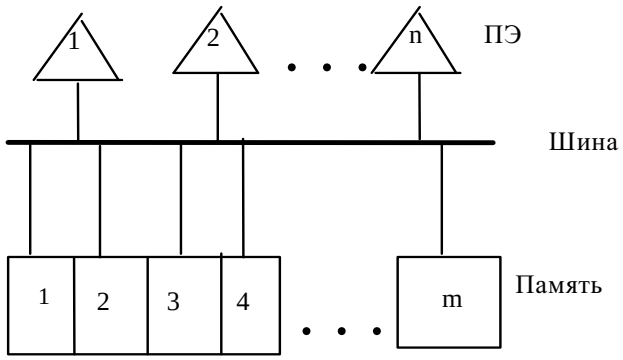
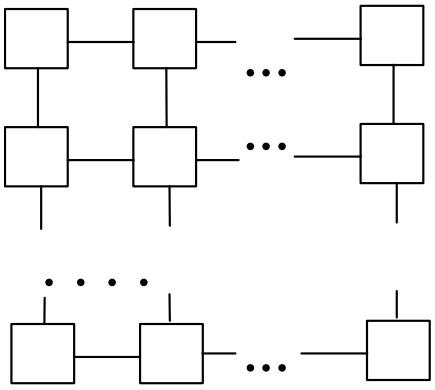


Рис. 3.1



Решетка ПЭ

Рис. 3.2

Понятно, что реализация доступа к разделяемой переменной существенно зависит от того, где размещаются взаимодействующие процессы. Если взаимодействующие процессы назначены на исполнение на один ПЭ, то доступ к

разделяемой переменной реализуется в программе процесса обычным образом - упоминанием имени переменной в программе, например, z:=x+1; . Если же взаимодействующие процессы назначены на исполнение в разные ПЭ, то доступ к разделяемой переменной реализуется в программе процесса через логический канал передачи данных.

3.2.Выполнение системы процессов

Проблема организации выполнения множества взаимодействующих процессов обсуждается в форме двух задач: *взаимное исключение* (*mutual exclusion*) и *производитель/потребитель* (*producer/consumer*).

Задача взаимного исключения формулируется следующим образом. Пусть выполняются два или более процессов, которым необходим доступ к одному и тому же неразделяемому ресурсу. Ресурс называется *разделяемым*, если несколько процессов одновременно могут его использовать, и *неразделяемым* в противном случае. Примером разделяемого ресурса служит переменная (вещественная переменная или файл), значение которой может одновременно считываться несколькими процессами. Процессы при этом не влияют друг на друга.

Эта же переменная является *неразделяемым* ресурсом при выполнении операции записи, поскольку запись новых данных влияет на выполнение других процессов, считывающих и использующих текущее значение переменной. Поэтому доступ к

переменной «по считыванию» могут одновременно иметь несколько процессов, но только один процесс может записывать новое значение переменной, запрещая при этом другим процессам и запись и считывание значения переменной. Если не запрещать считывание значения переменной в ходе выполнения записи нового значения, тогда результат считывания не будет однозначно определен (будет недетерминирован).

Взаимное исключение заключается в обеспечении доступа только одного процесса к неразделяемому ресурсу. Эта задача в действительности состоит из двух задач. Прежде всего, если несколько процессов p_1, p_2, ... , p_n требуют доступа к неразделяемому ресурсу, то только один из них должен быть допущен к ресурсу.

Выбор процесса, который допускается к ресурсу, составляет вторую задачу. Здесь возможна ситуация, когда один из процессов p_i, $i=1,2,...,n$, постоянно не выбирается и попадает в *вечное ожидание* затребованного ресурса (*starvation*). Алгоритм выбора процесса должен обеспечивать прогресс и не допускать вечного ожидания.

Задача производитель/потребитель заключается в следующем. Пусть есть пара процессов p_1 и p_2. Один из них - p_1 (производитель) - вырабатывает некоторый результат и помещает его в буфер v, а другой процесс - p_2 (потребитель) - должен результат считать. Для определенности будем предполагать, что

p_1 вырабатывает значение переменной и помещает его в буфер v, а p_2 считывает значение переменной из буфера v.

Если буфер v пуст, то потребитель должен ждать момента, когда новый результат будет произведен и помещен в буфер v.

Буфер может быть *ограниченным* или *неограниченным*. Ограниченный буфер может хранить конечное число значений (не более n, n - натуральное число) переменной. Такой буфер можно представить себе как очередь значений ограниченного размера. В неограниченном буфере всегда есть место для размещения следующего значения.

Если ограниченный буфер уже содержит n значений, то следующее значение не может быть в него помещено и процесс-производитель должен ждать момента, когда процесс-потребитель заберет очередное значение и освободит место для размещения нового значения.

Конечно, следует рассмотреть и случай нескольких процессов-производителей и нескольких процессов-потребителей.

Для анализа и решения этих задач необходимо прежде всего сформулировать и описать их в некотором формализованном виде. Для этой цели будут использованы сети Петри.

3.3.Сети Петри

Сеть Петри - это математическая модель, которая имеет широкое применение для описания поведения параллельных устройств и систем процессов. В настоящее время определены и изучены разнообразные классы сетей Петри и алгоритмов анализа их свойств [6-8]. Мы рассмотрим лишь самые общие понятия и возможности использования сетей Петри как для анализа названных задач, так и для задания прямого управления в параллельных программах. Наиболее интересны сети Петри тем, что они позволяют представлять и изучать в динамике поведение системы параллельных взаимодействующих процессов программы или любого другого дискретного устройства.

3.3.1.Определение сети Петри

Определение сети Петри дадим в три приема. *Сеть* есть двудольный ориентированный граф. Напомним, что двудольный граф - это такой граф, множество вершин которого разбивается на два подмножества и не существует дуги, соединяющей две вершины из одного подмножества. Итак, сеть - это набор

$G = (T,P,A),\ T \cap P = \varnothing,$

где

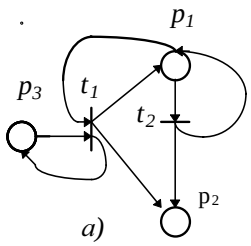
$T=\{t_1,t_2,...,t_n\}$-подмножество вершин, называющихся *переходы*,

P={p_1,p_2,...,p_m}-подмножество вершин, называющихся *местами*,

A⊆(T×P)∪(P×T) - множество ориентированных дуг.

По определению, ориентированная дуга соединяет либо место с переходом либо переход с местом.

П1. На рис. 3.3.а приведен пример сети в графическом представлении. Переходы обозначены черточками, а места - окружностями. Каждый переход *t* имеет набор входных *in{t}* и набор выходных *out{t}* дуг. Сети могут представляться также в форме продукционных правил (рис. 3.3.б).



а)

$t_1: \{p_3, p_1\} \rightarrow \{p_1, p_2, p_3\}$

$t_2: \{p_1\} \rightarrow \{p_1, p_2\}$

б)

Рис. 3.3.

3.3.2.Разметка сети

Сеть можно понимать (интерпретировать) по-разному. Можно представить себе, что места представляют условия (буфер пуст, значение переменной вычислено и записано в

буфер т.п.), а переходы - события (посылка сообщения в буфер, считывание значения переменной из буфера).

Состояние сети в каждый текущий момент определяется системой условий. Для того, чтобы стало возможным и удобным задавать условия типа “в буфере находится три значения” в определение сети Петри добавляются *фишки (размеченные сети)*. Фишки изображаются точками внутри места. В применении к программированию можно представлять себе переходы как процедуры, а места - как переменные.

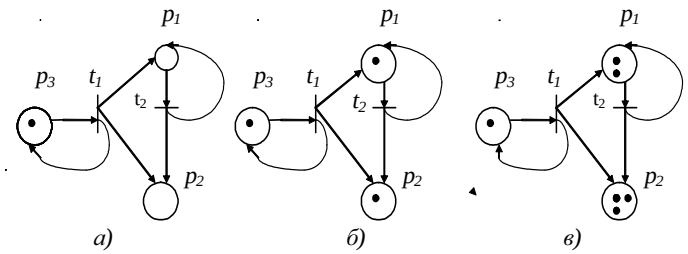


Рис. 3.4.

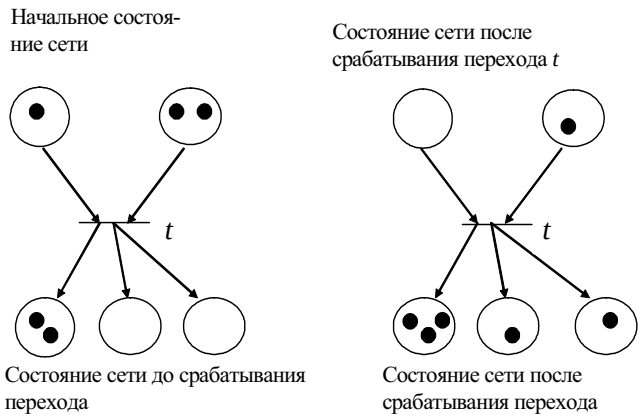


Рис. 3.5.

Фишка свидетельствует о том, что в переменной/буфере имеется значение, а если место имеет, к примеру, 3 фишки, то это может интерпретироваться как наличие трех значений в буфере. Если место содержит фишку, то место *маркировано* и сеть называется *маркированной*. Начальное распределение фишек задает начальную маркировку M_0 сети. Маркировка сети определяет ее текущее состояние.

Сеть на рис. 3.4 в начальном состоянии содержит одну фишку в месте p_3. Маркировка задается функцией M: $P \rightarrow I$, I={0,1,2,...}, а функция M представляется вектором, в котором i-ый компонент задает маркировку места p_i.

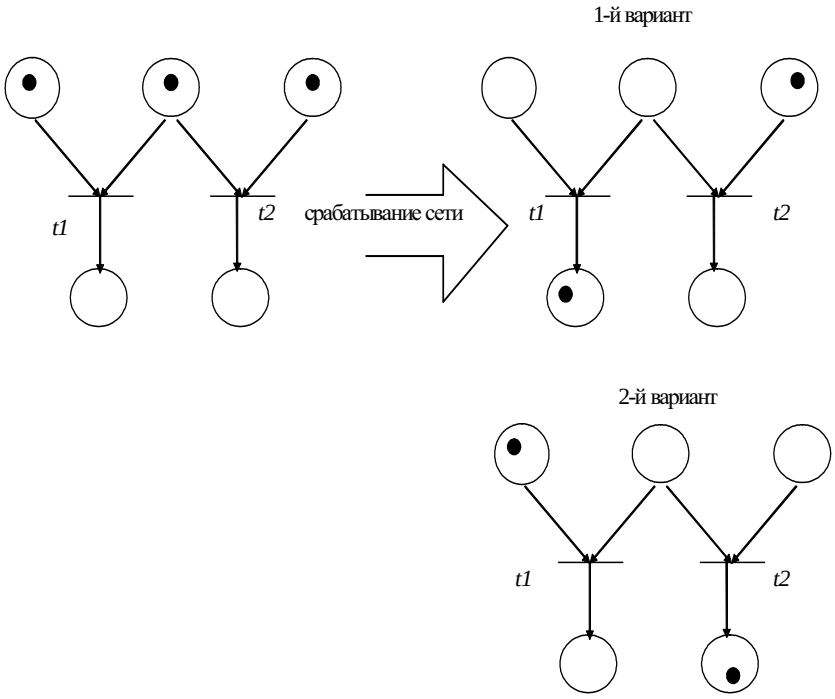


Рис. 3.6.

Например, начальная маркировка сети на рис. 3.4 представляется вектором M_0={1,0,0}.

И наконец в определение сети добавляется понятие *срабатывания* перехода. Срабатывание перехода состоит из того, что из всех входных мест перехода забирается по одной фишке и во все выходные места добавляется по одной фишке. Если представить себе переход как процедуру, то она корректно выполняется и вырабатывает значения своих выходных переменных, если есть значения всех аргументов. Таким образом,

переход может сработать, если хотя бы одна фишка находится в каждом из его входных мест, рис. 3.5. Сеть переходит из одного состояния в другое (от одной маркировки к другой) когда происходит событие - срабатывание перехода.

В другой интерпретации переход может представлять некоторое устройство. Устройство может - но не обязано! - сработать, если выполнились все входные условия. Если одновременно несколько переходов готовы сработать, то срабатывает один из них (любой), или некоторые из них, или все (рис. 3.6).

Итак, сетью Петри называется набор G=(T,P,A,M), $T \cap P = \varnothing$, где

$T=\{t_1,t_2,...,t_n\}$-подмножество вершин, называющихся *переходы*,

$P=\{p_1,p_2,...,p_m\}$-подмножество вершин, называющихся *местами*,

$A \subseteq (T \times P) \cup (P \times T)$ - множество ориентированных дуг.

M – функция $M: P \rightarrow I$, $I=\{0,1,2,...\}$.

Функционирование сети происходит от одного состояния к другому в результате срабатывания переходов.

На рис. 3.4 показана последовательность состояний сети Петри в ходе срабатывания переходов. Начальная разметка M_0 =(1,0,0) показана на рис. 3.4.а. В этом состоянии может сработать только переход t_1. Разметка сети M_1=(1,1,1) после срабатывания

t_1. показана на рис. 3.4.б. Разметка M_1=(1,1,1) позволяет одновременно сработать переходам t_1 и t_2, разметка M_2=(1,2,3) после их срабатывания показана на рис. 3.4.в.

3.3.3.Граф достижимости

Для использования сетей Петри необходимо знать их свойства, такие, например, как безопасность, а для этого следует изучить множество всех возможных разметок сети с заданной начальной разметкой.

Разметка *M* называется *достижимой*, если при некоторой конечной последовательности срабатываний переходов, начиная с начальной разметки M_0, сеть переходит к разметке *M*.

Граф достижимости определяет все достижимые разметки и последовательности срабатываний переходов, приводящих к ним (рис. 3.7). Его вершинами являются разметки, а дуга, помеченная символом перехода *t*, соединяет разметки M_1 и M_2 такие, что сеть переходит от разметки M_1 к разметке M_2 при срабатывании перехода *t*.

Любой конечный фрагмент графа достижимости, начинающийся с начальной разметки и до некоторых достижимых разметок называется *разверткой* сети. Множество всех разверток определяет *поведение* сети Петри. Пример различных разверток сети (элементы поведения [9]) показан на рис.3.7с. Каждая разметка *M* определяет *состояние* сети.

Состояние сети характеризуется множеством переходов, которые могут сработать в состоянии *M*.

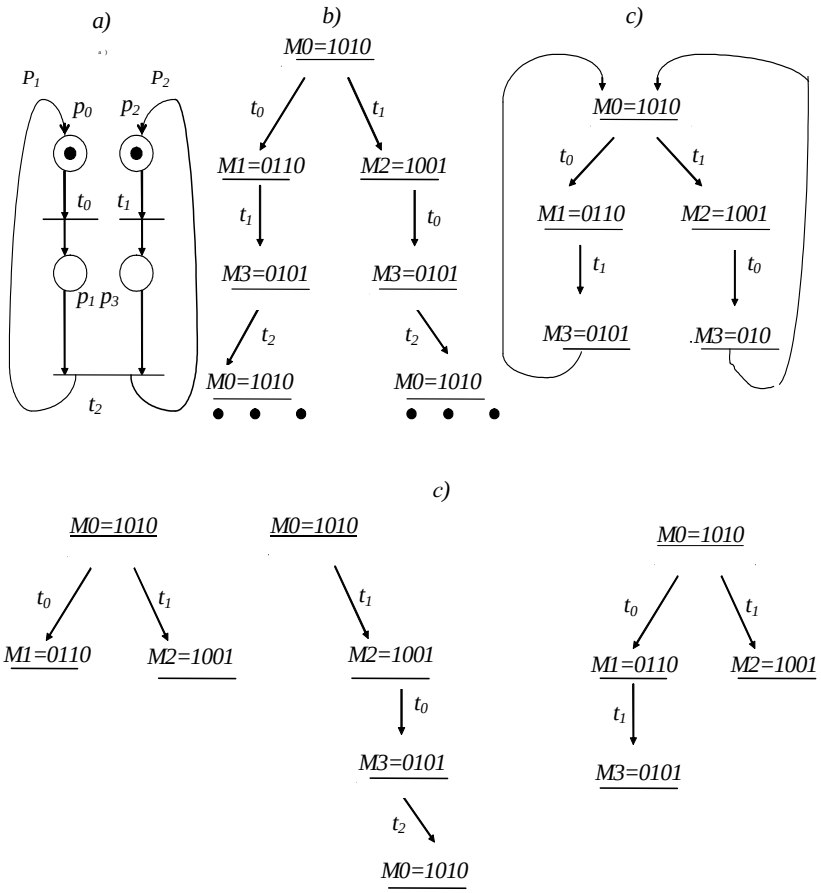


Рис. 3.7.

П2.Сеть на рис.3.7а. определяет управление двумя параллельно протекающими процессами с синхронизацией - оба процесса поставляют фишки, необходимые для срабатывания перехода t_2.

Граф достижимости показан на рис. 3.7б. Он бесконечен, однако после разметки M_3 в каждой ветви повторяется один и тот же фрагмент и потому возможно конечное представление графа достижимости (рис. 3.7.в). Множество разверток сети (ее поведение) бесконечно, примеры разверток приведены на рис. 3.7с.

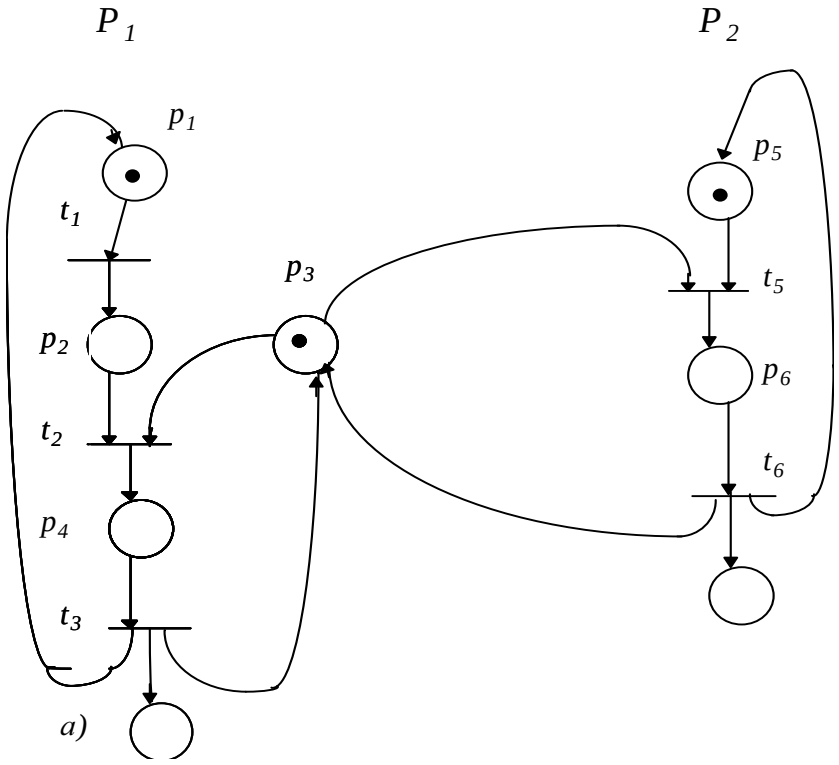
Вообще, всякое конечное дискретное устройство, вырабатывающее бесконечный результат, обнаруживает некоторого рода регулярность в поведении, что и обеспечивает конечную его представимость[1]. Это демонстрируют и примеры графов достижимости.

3.4.Задача взаимного исключения

Теперь сетью Петри можно строго описать поведение процессов в задаче взаимного исключения. Такая сеть должна описывать поведение системы процессов с взаимным исключением доступа к неразделяемому ресурсу.

Пусть заданы два процесса P_1 и P_2, конкурирующие за доступ к общему неразделяемому ресурсу (рис. 3.8). Общий ресурс изображается местом p_3. Переходы обозначают какие-то действия с использованием ресурсов. Например, если процессу P_1 выделен затребованный блок памяти p_3, то процесс P_1 сможет выполнить свою подпрограмму t_2. Количество экземпляров

ресурса p_3 (количество ресурса p_3) обозначается фишками в месте p_3 - один экземпляр. Итак, два процесса (переходы t_3 и t_5) запрашивают единственный экземпляр ресурса p_3 но только одному процессу ресурс может быть выделен. Во множестве *поведение* сети на рис 3.8 нет такой развертки сети, которая бы привела к одновременному срабатыванию переходов t_3 и t_5.



[1] В теории формальных языков, например, этот факт выражается в

Рис. 3.8.

Сеть Петри не определяет, какой именно из двух конкурирующих процессов получит доступ к ресурсу, она лишь описывает ограничение на доступ к ресурсу - только один процесс (все равно какой) получает доступ к ресурсу p_3. Как следствие, при таком задании управления возможна ситуация, когда доступ к ресурсу будет постоянно получать один и тот же процесс, например P_1, а процесс P_2 останется “навечно” ожидать выделения ему ресурса p_3 (состояние *starvation*).

При организации выполнения системы процессов эта ситуация разрешается с использованием дополнительных средств. Например, в операционных системах в описание состояния процессов вводится дополнительная характеристика - *приоритет*. Запрошенный ресурс выделяется процессу с наибольшим приоритетом. Начальный приоритет любого процесса растет с ростом времени ожидания ресурса. Таким образом, всякий процесс со временем получит запрошенный ресурс.

Другой (наиболее распространенный) способ выделения ресурса - устройство очереди. Все запросы ресурса выстраиваются в очередь к ресурсу и процесс, раньше всех

---

pumping теоремах.

запросивший ресурс, получит его первым (дисциплина обслуживания FIFO - First_In - First_Out).

3.5.Дедлоки

Сеть Петри оказалась удобным средством для анализа такого свойства параллельной системы процессов как наличие или отсутствие *дедлоков*. Система взаимодействующих и конкурирующих за ресурсы процессов попадает в состояние *дедлока* вследствие ошибок в управлении, при которых даже при наличии достаточных для выполнения программы ресурсов система процессов не может удовлетворить все запросы ресурсов.

3.5.1.Определение дедлока.

Если запрос ресурсов в системе не может быть удовлетворен, то система останавливается (ни один переход не может сработать). Это может быть нормальный останов сети, а может быть следствие конкуренции за ресурсы.

Рассмотрим внимательно пример на рис. 3.9.а. Сеть *A* определяет циклическое неограниченное срабатывание переходов t_1, t_2, t_3 (процесс *A*). При срабатывании переходы t_2 и t_3 потребляют единицу ресурса из мест p_3 и p_7 каждый. Можно представить себе для определенности, что места p_3 и p_7 обозначают какие-то внешние устройства. Пока процесс, управляемый сетью *A*, выполняется один, ситуации дедлока не

возникает. Но если появляется другой процесс (рис. 3.9.б), выполняющийся параллельно *A* и управляемый сетью *B* (процесс *B*), тогда появляется конкуренция за ресурс в местах p_3 и p_7.

Состояние дедлока возникает при следующей последовательности срабатываний переходов сети: t_1, t_4, t_2, t_5. Теперь имеем дедлок: все ресурсы потреблены, новый запрос не может быть удовлетворен и ни один переход не может сработать. Однако сеть будет нормально функционировать, если в месте p_1 оставить одну фишку, т.е. разрешить выполняться либо процессу *A* либо процессу *B*, но не обоим одновременно.

Основную опасность при таком динамическом (в ходе вычислений) захвате ресурсов вызывает *дозахват* ресурса (запрос дополнительной порции того же или другого ресурса без освобождения уже захваченных ресурсов). Система процессов в состоянии дедлока “зависает” в ожидании и не может никогда завершиться. Следовательно, необходимо разработать стратегии для борьбы с дедлоками.

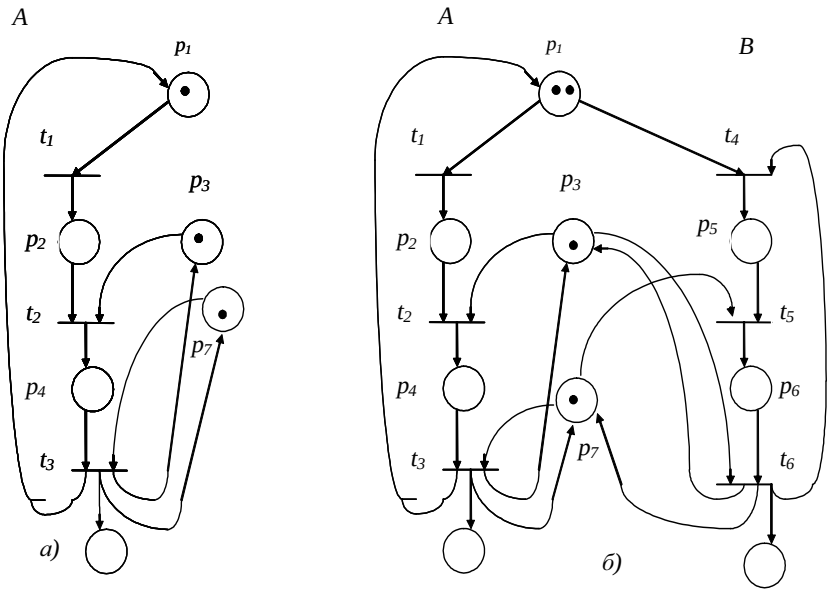


Рис. 3.9.

3.5.2.Необходимые условия возникновения дедлока.

Рассмотрим детально условия, которые привели к возникновению состояния дедлока.

Взаимное исключение. Для правильного исполнения процессов *A* и *B* необходимо было организовать их взаимное исключение, так как ресурсы p_3 и p_7 являются неразделяемыми ресурсами. Если процесс *A* получил все экземпляры ресурса p_3, то другой процесс, затребовавший этот же ресурс должен быть задержан и ждать освобождения необходимого количества ресурса p_3.

Дозахват ресурса. Каждый процесс получил часть ресурса p_3, держит его (не освобождают) и еще затребовал дополнительный ресурс p_7.

Не допускается временное освобождение ресурса. Дедлок не возник бы, если бы была возможность:

- задержать один из процессов (например A) до его завершения,
- временно освободить принадлежащий A ресурс (*preemption*) p_3,
- отдать ресурс p_3 для завершения процесса B,
- после завершения процесса B продолжить исполнение A уже со всеми необходимыми ресурсами.

Некоторые ресурсы допускают такое освобождение, но не все. Например, если p_3 и p_7 - участки памяти, то для завершения процесса B операционная система может эвакуировать процесс A во вторичную память (*swapping*), отдать освободившийся участок памяти p_7 процессу B и после его завершения передать освободившиеся ресурсы для завершения A.

Если p_3 - принтер, на который процесс A уже начал вывод информации, то освободить его на время и передать процессу B невозможно, если, конечно, не планируется специально напечатать смесь сообщений из обоих процессов.

Циклическое ожидание. Дедлок на рисунке 3.9 демонстрирует циклическое ожидание: процесс *A* получил ресурс p_3 и ожидает получения ресурса p_7. А процесс *B* имеет ресурс p_7 и ожидает получения ресурса p_3. Для анализа распределения ресурсов рассматривается *граф распределения ресурсов* (ГРР). ГРР - это двудольный граф, его вершины - имена ресурсов и процессов. Дуга ведет из вершины-ресурса *r* в вершину-процесс *p*, если ресурс *r* выделен процессу *p*. Дуга ведет из вершины-процесса *p* в вершину-ресурс *r* , если процесс *p* запросил ресурс *r*, но еще не получил его и находится в состоянии ожидания. Система процессов на рис.3.9 в состоянии дедлока характеризуется ГРР на рис.3.10. Как обычно, фишки показывают количество ресурсов. На графе явно видно циклическое ожидание процессов. Говорят, что процессы *A* и *B* *входят в цикл ожидания*. Понятно, что такой цикл могут образовать и более чем два процесса и ресурса.

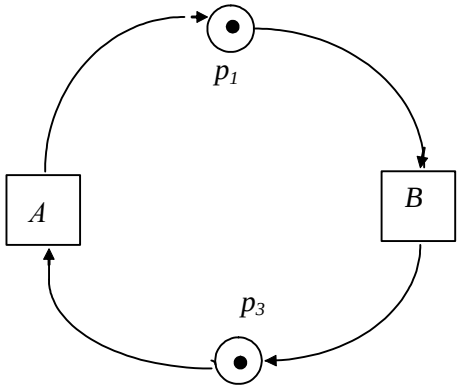


Рис. 3.10

3.5.3.Борьба с дедлоками.

Возможно два подхода к борьбе с дедлоками. Первый основан на таком аккуратном планировании ресурсов, при котором дедлок заведомо не может возникнуть - *предотвращение дедлоков*. Второй подход допускает дедлоки, но есть алгоритмы обнаружения и выхода из состояния дедлока - *преодоление* дедлока. Реализация обоих подходов оказалась весьма дорогостоящей.

Предотвращение дедлоков. Условия 1-4 составляют необходимое, но не достаточное условие возникновения дедлока. Все эти условия могут удовлетворяться, но дедлок не обязательно возникает. Однако если нарушено хотя бы одно из условий 1-4, то дедлок вообще не возникнет. На этом основано предотвращение дедлоков. Посмотрим, каким образом можно избежать удовлетворения условий 1-4.

Первое условие - взаимное исключение - нарушить не удастся, если распределяется неразделяемый ресурс. Тут ничего поделать нельзя.

Второе условие - дозахват - нарушить можно. Одна возможность используется операционными системами: процесс требует все необходимые ресурсы изначально и начинает исполняться лишь тогда, когда получит все, что запросил. Часть

ресурсов при этом может долго не использоваться процессом. Неиспользуемые, но захваченные, ресурсы, а также начальное ожидание процессов, пока для них будут выделены все необходимые ресурсы, и есть цена безопасности системы взаимодействующих процессов. Например, если мультикомпьютер состоит из 500 процессоров и параллельная программа запросила для своего исполнения 200 процессоров, то такая программа при хорошей загрузке мультикомпьютера может стоять в очереди сутками, ожидая, когда освободится разом 200 процессоров. Либо оператор должен остановить продвижение очереди программ, не распределять освободившиеся процессы ожидающим программам и копить 200 свободных процессоров – тоже очень накладно.

Другой способ заключается в том, что процессу разрешается запрашивать новый ресурс лишь тогда, когда процесс не имеет никаких ресурсов вовсе. Следовательно, процесс должен сначала отказаться от всех своих ресурсов и лишь тогда снова сможет запросить все нужные ресурсы.

Временное освобождение ресурса (если это возможно) должно специально обеспечиваться ОС и оборудованием вычислителя так, как в современных ЭВМ обеспечивается временная приостановка выполнения программы и её эвакуация во вторичную память (поддерживается системой прерывания, страничной организацией памяти и т.п.) для освобождения

оперативной памяти, понадобившейся для завершения другой, более приоритетной программы.

Если неразделяемый ресурс не может быть временно освобожден, то в ходе выполнения процесса он может иногда заменяться на другой, но уже разделяемый ресурс (а потому не нужно обеспечивать взаимное исключение), а реально необходимый неразделяемый ресурс может быть позже запрошен на короткое время в наиболее удобный момент. Примером может служить техника *спулинга* (spooling), при которой, к примеру, принтер (его временное освобождение невозможно) заменяется дисковым файлом печати. Диск не назначается ни одному процессу в монопольное использование. В файле печати накапливается весь материал для печати и затем реальный принтер используется кратковременно в удобное для системы распределения ресурсов время (когда процесс освободил, к примеру, все свои ресурсы) для “залповой” печати.

Другие подходы к предотвращению дедлоков требуют использования дополнительной информации. Полезнее и проще всего изначально знать максимальное количество ресурсов, которое может запросить процесс в ходе выполнения. Эта информация позволяет всегда из множества готовых процессов выбирать на исполнение такое подмножество процессов, которое не попадает в состояние дедлока. Идея такого планирования

формулируется в виде *алгоритма банкира*, суть которого следующая.

Пусть в вычислительной системе есть n типов ресурсов, каждого типа ресурсов - r_i экземпляров. Одновременно исполняются m процессов $P_1, P_2, \ldots, P_m$. Каждый процесс P_j, $j=1,\ldots,m$, максимально может затребовать k^j_1 экземпляров ресурса первого типа, k^j_2 - экземпляров ресурса второго типа, k^j_n - экземпляров ресурса n-го типа. Для каждого j-го ресурса и процессов $P_1, P_2, \ldots, P_m$ должно выполняться условие банкира:

$$\sum_i k^i_j \le r_j$$

Другими словами, суммарный максимальный запрос всех процессов $P_1, P_2, \ldots, P_m$ любого j-го ресурса не превосходит его наличного количества r_j. Следовательно, все запросы процессов могут быть удовлетворены. Каждому новому процессу разрешается начать исполнение лишь в том случае, если он не нарушит условие банкира.

Преодоление дедлока. Если вычислительная система допускает состояние дедлока, то его необходимо уметь обнаружить и уметь выйти из этого состояния. Дедлок обнаруживается, если выполняются необходимые условия дедлока и нарушено условие банкира. Такая проверка должна делаться систематически.

При обнаружении дедлока необходимо разорвать циклическое ожидание, которое находится из анализа ГРР. Для

этого можно принести в жертву один из процессов, входящих в цикл ожидания, и “убить” его. Если циклическое ожидание еще не разорвано, тогда необходимо “убить” следующий процесс и так до тех пор, пока система процессов не выйдет из состояния дедлока. Конечно, выбрать очередную “жертву” невозможно без привлечения дополнительной информации о процессах и хорошего решения, как правило, найти не удается.

3.6.Задача о пяти обедающих философах

Рассмотрим классическую задачу о пяти обедающих философах. Суть задачи такова. Пять философов, прогуливаясь и размышляя, время от времени испытывают приступы голода. Тогда они заходят в столовую, где стоит круглый стол, на нем всегда приготовлены пять блюд. Между соседними блюдами лежит одна вилка (всего лежит ровно пять вилок). Голодный философ (рис. 3.10):

а) входит в столовую, садится за стол и берет вилку слева,

б) берет вилку справа,

в) ест (обязательно двумя вилками),

г) кладет обе вилки на стол, выходит из столовой и продолжает думать.

При конструировании управления в этой задаче следует учитывать самые разнообразные варианты поведения философов. Рассмотрим некоторые из них.

а).Необходимо организовать действия философов так, чтобы они все были накормлены и не случилось бы так, что пять философов одновременно войдут в столовую, возьмут левую вилку и застынут в ожидании освобождения правой вилки (дедлок!). Голодная смерть всех философов неминуема, если никто из них не пожелает расстаться на время со своей левой вилкой. Будет не лучше, если они одновременно положат левые вилки, а затем вновь одновременно попытаются завладеть необходимыми двумя вилками. Результат, понятно, тот же. Типичный дедлок в результате попытки дозахватить ресурс (вторую вилку)! Выход из него - “убить” один из процессов (выставить за дверь одного, все равно какого, голодного философа и никого более в столовую пока не пускать). Однако гарантировать, что все философы будут накормлены и ни один из них не попадет в состояние вечного ожидания в этой стратегии нельзя.

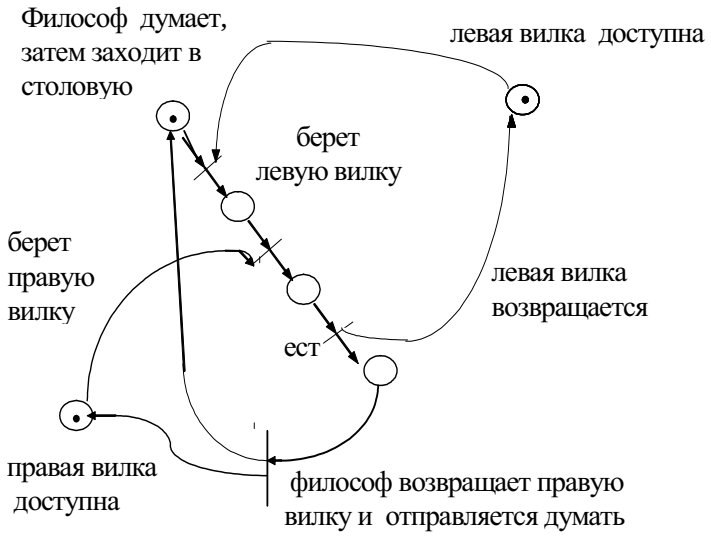


Рис. 3.10

Очевидно отсюда, что грубым (не эффективным) приемом, чтобы избежать этой ситуации, является введение ограничения на число философов, допущенных одновременно в столовую. Например, можно пускать в столовую не более четырех философов одновременно. Тогда, по крайней мере, один из них сможет захватить две вилки, поесть и освободить ресурсы (вилки) для других.

б).Необходимо также предусмотреть, чтобы два философа одновременно не хватали одну и ту же вилку (обеспечить взаимное исключение). Зная вспыльчивый характер философов, нетрудно в этом случае предсказать результат.

в).Стеснительный философ не должен умереть в столовой голодной смертью из-за того, что его вилки постоянно раньше него хватают более напористые соседи (не допустить состояния вечного ожидания).

г).Легко представить себе ситуацию, когда банда сговорившихся философов завладеет всеми вилками и, передавая их только в своей среде, уморит голодом всех прочих.

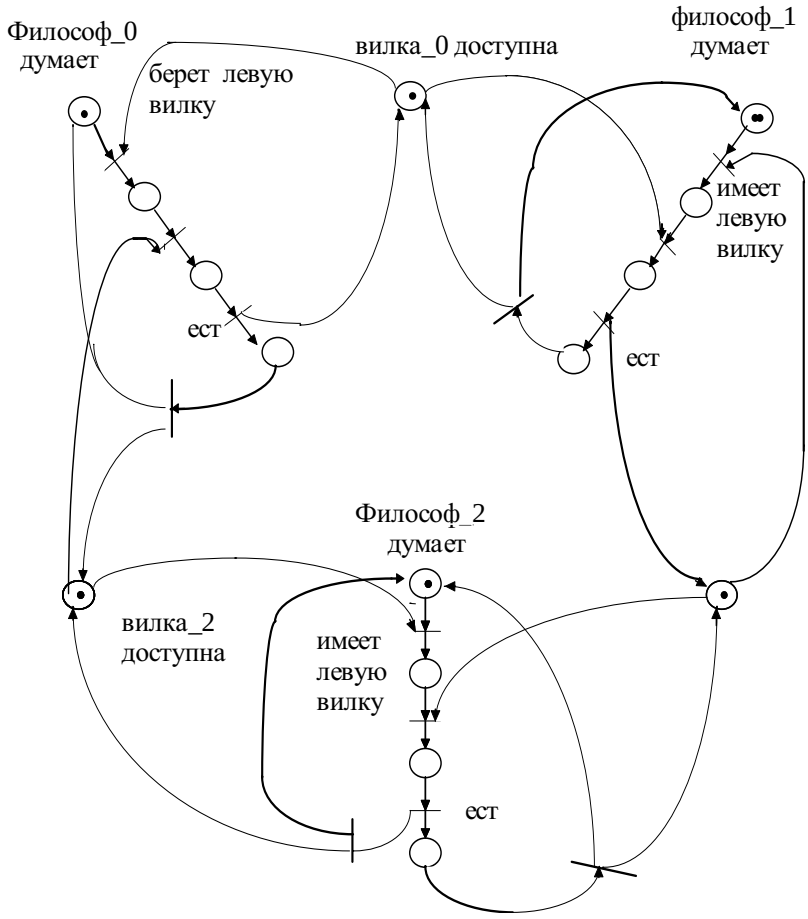


Рис. 3.11.

На рис. 3.11 представлена сеть Петри, задающее управление распределением вилок трем обедающим философам, человеческий фактор в ней не учитывается (пункты б-г). Эта сеть конструировалась пошагово.

Вначале была сконструирована сеть, управляющая поведением одного (любого) философа (рис. 3.10). Затем три таких сети, соответствующие трем философам, объединяются в одну. Распределяемые ресурсы - левые и правые вилки - разных фрагментов совмещаются.

В качестве упражнения можно попробовать сконструировать сеть Петри, которая бы управляла пятью философами так, что ни один из них не умрет голодной смертью (всем будет обеспечен равный доступ в столовую).

3.7.Задача производитель/потребитель

Поведение процессов производителя и потребителя описано сетью примера на рис. 3.12.а. Здесь производитель П производит детали и оставляет их на складе, а потребитель Т забирает их со склада, когда они там есть. Регулирует это взаимодействие место p_5. Склад здесь описан неограниченно большой емкости. Процесс-производитель может сработать неограниченно большое число раз, даже если процесс потребитель не работает совсем.

Если необходимо принять во внимание ограниченную вместимость склада, тогда в сеть добавляется место p_6 (рис. 3.12.б). Оно определяет наличие места для хранения не более чем 10 деталей. Взятие фишки из места p_6 может интерпретироваться как взятие разрешения поместить очередную деталь на склад (занять свободное место для хранения).

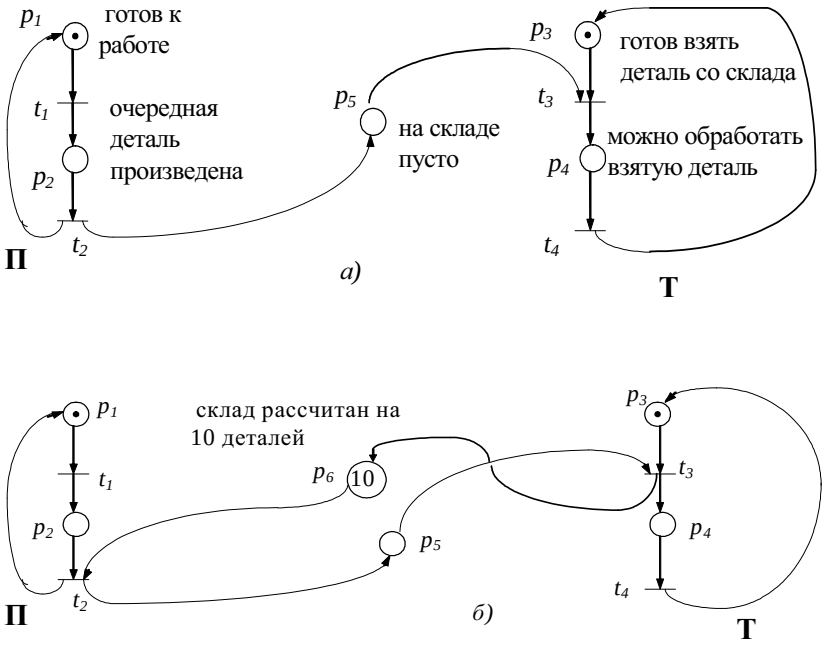


Рис. 3.12.

ПЗ.Рассмотрим пример конвейера. Пусть есть три обрабатывающих устройства t_0, t_1, t_2, организованные в виде конвейера. Это могут быть, например, станки на заводе или функциональные устройства конвейерного процессора и вообще любой конвейер, в котором каждое обрабатывающее устройство выполняет лишь часть общей работы, а конечный результат будет выработан последним из них.

Пусть требуется, чтобы места p_1 и p_2 могли содержать ограниченное число результатов: место p_1 может вместить лишь два результата (место p_1 сети *2-ограничено*) предшествующего

этапа работы конвейера (вырабатывается переходом t_0), а место p_2 – 3-ограничено. Символ n в месте p_0 означает наличие n фишек в нем, n - целое положительной число.

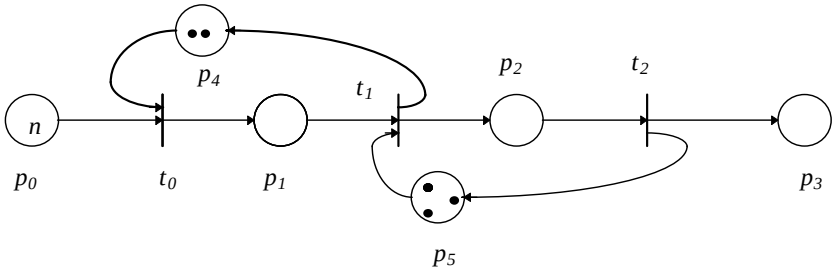


Рис. 3.13.

Сеть Петри, обеспечивающая необходимое прямое управление, приведена на рис. 3.13. Понятно, что в месте p_1 не может накопиться более 2-х фишек при любых порядках срабатывания переходов сети. Места p_1 и p_2 еще называют *асинхронными каналами*, с их помощью реализуется программирование средствами асинхронного *message passing interface*.

Сеть Петри, в которой все места 1-ограничены, называется *безопасной*. Такой сетью можно задавать прямое управления в программах. Безопасная сеть никогда не допустит, чтобы в переменную было положено новое значение, если старое еще не было использовано по назначению. Нарушения этого

правила может быть причиной ошибок в параллельных программах.

При использовании сетей Петри в языках программирования стандартные схемы управления (например, сеть управления конвейерным исполнением процедур) могут быть описаны как управляющие процедуры, например:

control procedure *pipiline* (t_0, p_4, t_1, p_5, t_2);

<описание сети примера **П4**>;

Теперь при необходимости задать в программе конвейерное вычисление некоторых процедур, их имена подставляются в обращение к управляющей процедуре *pipiline* вместо формальных параметров t_0, t_1, t_2.

call pipiline (t_0=*proc1*, p_4=2, t_1=*proc2*, p_5=3, t_2=*proc3*)

Наличие библиотеки стандартных управляющих процедур способно значительно облегчить отладку взаимодействия процессов.

Сеть примера **П4** может быть использована также для управления асинхронным каналом при описания и реализации message passing interface в языках с асинхронными взаимодействиями.

Сети Петри удобны для задания прямого управления в теоретических работах при исследовании параллелизма. К

сожалению оказалось, что их трудно использовать в языках программирования как в силу большой времяемкости моделирования их поведения, так и в силу сложности конструирования и отладки заданного ими прямого управления.

3.8.Реализация управления взаимодействующими процессами

Сети Петри описывали поведение процессов, но не определяли, как это поведение реализовать (запрограммировать). В языках программирования для этого разработаны многочисленные средства, в основе которых лежат операции над семафорами.

3.8.1.Семафоры

Семафоры являются тем базовым средством, с помощью которого можно реализовать в программе управление параллельно протекающими процессами, их синхронизацию и взаимодействия.

Семафор S есть переменная типа **integer**, которая после инициализации начального значения доступна только посредством семафорных операций *P* (от голландского слова *proberen* - проверить) и *V* (от голландского слова *verhogen* - увеличить, прирастить). Никаким другим способом значение семафорной переменной не может быть ни проверено, ни изменено.

Операции *P* и *V* определяются следующим образом:

P(S): **while** $S \leq 0$ **do** *skip;*

S:=S-1;

V(S): *S:=S+1*;

Каждая из операций *P* и *V* является неделимой, т.е. если семафорная переменная изменяется одной из операций, то в это время к ней нет доступа ни для какого процесса. Таким образом, изменение значения семафорной переменной (*S:=S-1* или *S:=S+1*) может делаться только одной операцией (одним процессом).

В определении семафорной операции *P* оператор цикла

while $S \leq 0$ **do** *skip;*

описывает алгоритм выполнения операции *P(S)*. В реализации операции *P* нет, конечно, необходимости реально циклить на этом операторе. Обычно операция *P(S)* реализуется таким образом, что если в процессе при выполнении операции *P(S)* удовлетворяется условие $S \leq 0$, то процесс переводится в состояние ожидания и вновь инициируется только по выполнению операции *V(S)* в любом из процессов.

Понятно, что при доступе к семафору тоже возможна ситуация вечного ожидания - один из процессов постоянно не получает разрешения завершить операцию *P(S)*.

Неделимое исполнение семафорных операций в мультипроцессорах с разделяемой памятью (все процессоры работают над общей памятью) обеспечивается специальной машинной инструкцией “Проверить и изменить”. Оборудование допускает исполнение этой инструкции только одним

процессором (и, следовательно, только в одном процессе). Все остальные процессоры задерживаются на время ее исполнения.

В мультикомпьютерах семафорные операции реализуются программным обеспечением.

3.8.2. Задача взаимного исключения

Взаимное исключение может быть реализовано с помощью семафоров следующим образом.

Пусть для *n* процессов *Proc(i), i=1,2, ... ,n,* должно быть обеспечено взаимное исключение при доступе к некоторому ресурсу (все равно какому). Для программирования взаимного исключения используется семафорная переменная *mutex* (mutual exclusion). Тогда структура программы *i*-го процесса такова:

var *mutex=1*: **semaphore;**

/* семафорная переменная *mutex* инициируется со значением 1 */

Proc(i) :

repeat

. . . /* начальная часть программы процесса*/

P(mutex)

. . . /*критический интервал*/

V(mutex)

. . . /* заключительная часть программы процесса */

until *false*

Программу процесса составляют операторы языка программирования, заключенные между операторами **repeat** и **until**. Логически программа процесса делится на три части. Участок программы процесса между операторами *P(mutex)* и *V(mutex)* называется *критическим интервалом*. Здесь выполняются те вычисления, ради которых затевалось взаимное исключение. Критический интервал “охраняется” семафором *mutex* от влияния других процессов. Действительно, если инициализировать семафорную переменную *mutex* значением 1, то только один процесс будет в состоянии выполнить операцию *P* и войти в свой критический интервал. Все остальные процессы будут ожидать на операторе

while *mutex* ≤ 0 **do** *skip;*

Завершив выполнение критического интервала, процесс выполнит оператор *V(mutex)* и увеличит значение семафорной переменной *mutex* на единицу. Следовательно, один из ожидающих процессов (неизвестно какой) получит право войти в свой критический интервал.

Если инициировать семафорную переменную со значением, например, 3, то тогда 3 процесса получат право одновременно выполнять свои критические интервалы.

3.8.3. Задача производитель/потребитель с ограниченным буфером

Накопительный буфер имеет два пограничных состояния, ограничивающих активность процессов-потребителей и процессов-производителей:

буфер пуст - процессы-потребители должны ждать

и

буфер полон - процессы-производители должны ждать.

Заведем соответственно два семафора для описания этих состояний:

b_empty - содержит количество свободных позиций для размещения новых элементов данных в буфере, инициализируется значением *n*,

и

b_full - содержит количество элементов данных в буфере, инициализируется значением 0.

Еще один семафор - *mutex* - обеспечивает взаимное исключение процессов.

var *b_empty=n, b_full=0, mutex=1*: **semaphore; . . .**

var *full=0, empty=n; /*счетчики элементов и свободных мест*

Producer||Consumer;

/* оператор || разрешает параллельное исполнение процессов *Producer* и *Consumer* */

Producer:

```
        {
        repeat
                ...
              производится очередной элемент данных
                ...
             P(b_empty); /*число работающих процессов-
                          производителей не должно
                          превышать числа свободных мест в
                          буфере*/
              P(mutex);
                ...
добавляется вновь произведенный элемент данных  в буфер
                ...
              V(mutex);
              V(b_full);
        until false;
        };
Consumer:
        {
        repeat
              P(b_full); /*число работающих процессов-
                        потребителей не должно превышать
                        числа произведенных элементов в
                        буфере*/
```

P(mutex);

...

элемент данных забирается из буфера

...

V(mutex);

V(b_empty);

...

until *false;*

}

3.8.4. Задача читатели-писатели

Рассмотрим более сложный пример решения задачи программирования взаимодействия множества процессов с использованием семафоров. Пусть выполняются *n* процессов, которые разделяют некоторую переменную программы. Часть процессов - *писатели* - только модифицируют разделяемый объект, другие - *читатели* - только считывают значение. Необходимо организовать корректное выполнение этой системы взаимодействующих процессов.

Прежде всего отметим, что процессы-читатели могут иметь одновременный доступ к разделяемому объекту, т.к. чтение не меняет значение объекта и, следовательно, процессы-читатели не могут влиять друг на друга. Для процессов-

писателей следует организовать взаимное исключение при доступе к разделяемому объекту.

Далее, процессы-писатели должны иметь преимущество при доступе к разделяемому объекту, поскольку они готовы записать новые данные. Следовательно, процессы-читатели должны получать доступ к разделяемому объекту лишь в том случае, если нет процессов-писателей, желающих получить доступ к этому разделяемому объекту. А потому, если процесс-писатель изъявил желание получить доступ к разделяемому объекту, то все процессы-читатели, которые в этот момент времени тоже желают получить доступ к разделяемому объекту, должны быть задержаны. С другой стороны, процессы-читатели нехорошо совершенно дискриминировать и потому процессы-читатели, изъявившие желание получить доступ к разделяемому объекту до того, как появился процесс-писатель, должны иметь возможность завершить чтение.

Эта модельная задача может реально интерпретироваться. Такая схема взаимодействий должна быть реализована, если нужно, к примеру, сделать Интернет-газету, которую любой читатель может читать, а репортеры (процессы-писатели) могут в любой момент, по мере поступления новой информации, её обновлять

Для программирования такого взаимодействия будут использованы 4 семафорные переменные *Mutex_n_writers*,

Mutex_n_readers, *NoWriter* и No*Reader* и счетчики процессов-читателей *n_readers* и процессов-писателей n_writers[1].

<table>
<tr><td colspan="2">/* Вначале описываются общие (глобальные) переменные множества процессов*/
var Mutex_n_writers:=1, Mutex_n_readers:=1, NoWriters:=1, NoReaders:=1 semaphore;
var n_readers:=0, n_writers:=0 integer;</td></tr>
<tr><td>Writer: {</td><td>Reader: {</td></tr>
<tr><td colspan="2">/*Если стартовал хотя бы один процесс-писатель, то первым делом следует позаботиться о блокировании всех процессов-читателей, которые стартуют после него */</td></tr>
<tr><td>P(Mutex_n_writers);
n_writers++;
/*стартовал очередной процесс-писатель*/
V(Mutex_n_writers);
/*произвольные действия, не затрагивающие разделяемые переменные*/</td><td>var wait bool;
do {
P(NoWriters);
wait = n_writers!=0;
if (wait) V(NoWriters);
} while (wait);
P(Mutex_n_readers);
n_readers++;
if (n_readers==1)
P(NoReaders);</td></tr>
</table>

[1] Программу разработал магистрант НГТУ А.Уразов

	V(Mutex_n_readers); *V(NoWriters)*;
/*Если нет работающих процессов-писателей, то все вновь стартовавшие процессы-читатели получают беспрепятственный доступ к разделяемому ресурсу. А все позднее стартовавшие процессы-писатели должны подождать завершения уже работающих процессов-читателей	
/*Теперь процессы могут работать по своим программам и в некоторый момент придут в точку доступа к разделяемому ресурсу*/	
P(NoWriters); … … *P*(*NoReaders);* /* осуществляется запись */ *V*(*NoReaders)*; *V(NoWriters)*;	. . . /* осуществляется чтение */
/* Далее может идти некоторый другой фрагмент программы процесса и наконец его завершающая часть*/	
P(Mutex_n_writers); *n_writers--*; /*процесс-писатель	*P(Mutex_n_readers);* *n_readers--;* **if** *(n_readers==0)*

завершился*/ *V(Mutex_n_writers);* **}**	*V(NoReaders);* *V(Mutex_n_readers);* /*если завершился последний процесс-читатель, то надо разрешить работать процессам-писателям*/ **}**

В программе не рассматриваются случаи аварийного завершения. Например, если последний процесс-читатель или последний процесс-писатель завершится аварийно и не откроет барьерный семафор (*V(Mutex_n_writers) и/или V(Mutex_n_readers)*), тогда система процессов, конечно, «зависнет».

3.8.5. Критические интервалы

Программируя межпроцессные взаимодействия с использованием семафоров легко допустить разнообразные неочевидные ошибки, в первую очередь ошибки временные, связанные с доступом к разделяемым данным.

В языках программирования вместо семафоров используются другие конструкции более высокого уровня. Одна из них - критические интервалы. Ее идея такова.

Вводится понятие *разделяемой* переменной, которая доступна из нескольких процессов. Например:

var *x* : **shared real;**

Разделяемые переменные доступны только внутри оператора **region** вида

region *x* **do** *S*;

Только один процесс может исполнять оператор **region** с переменной *x* в качестве параметра. Таким образом, пока исполняется оператор *S* никакой другой процесс не может начать исполнение оператора **region** *x*. Понятно, что оператор **region** *x* реализуется программой

var *x* : **semaphore**;

P(x)

S;

V(x)

Программировать с использованием оператора критического интервала легче, однако в дедлок тоже легко попасть, например:

P1: **region** *x* **do** (**region** *y* **do** *S1*);

P2: **region** *y* **do** (**region** *x* **do** *S2*);

Очевидна возможность дедлока как следствие дозахвата ресурса, когда два процесса *P1* и *P2* одновременно начнут выполнять свои вложенные операторы **region**.

IV. ПРОГРАММИРОВАНИЕ ВЗАИМОДЕЙСТВУЮЩИХ ПРОЦЕССОВ

Ниже приведены несколько примеров программирования взаимодействия параллельно исполняющихся процессов. Все они определяют асинхронное взаимодействие процессов.

4.1.Асинхронное программирование

Асинхронная модель вычислений наиболее пригодна для описания вычислений на мультипроцессоре и/или мультикомпьютере. Асинхронная программа определяет систему независимо исполняющихся и взаимодействующих процессов. Существует много различных воплощений этой модели вычислений в различных языках программирования. Здесь эта модель рассматривается в самом общем виде.

4.1.1.Понятие асинхронной программы

Асинхронная программа (А-программа) - это конечное множество А-блоков $\{A_k \mid k \in \{1,2,...,m\}\}$ определенных над информационной IM и управляющей CM памятями.

Каждый А-блок A_k=<*tr(ak),ak,c(ak)*> состоит из *спусковой* функции *tr(ak) (trigger function),* операции *ak, ak*$\in$F, и *управляющего* оператора *c(ak).*

Памятью называется конечное множество ячеек памяти, способных хранить значения переменных.

Спусковая функция *tr(ak)* - это предикат, в программе обычно задается условным выражением либо логической функцией. Как обычно здесь, операция *ak* реализуется в программе одноименной процедурой, вычисляющей функцию f_a.

Управляющий оператор *c(ak)* - оператор присваивания или процедура, меняющая значения ячеек управляющей памяти CM (например, чтобы разрешить или запретить исполнение какого-нибудь А-блока).

Каждой переменной из in(*ak*)$\cup$out(*ak*) в памяти IM соответствует ячейка, в которой хранится ее значение.

Выполнение А-программы состоит:

- в вычислении значения спусковой функции *tr(ak)* для всех или части А-блоков,
- в выполнении одного, части или всех А-блоков A_k таких, что *tr(ak)*=**true** при текущем состоянии памяти IM$\cup$CM.

Таким образом, исполняющая система в зависимости от имеющихся в наличии свободных на этот момент ресурсов имеет право инициировать любое подмножество А-блоков, готовых к исполнению (с истинной *tr(ak*)). Понятно, что при разных исполнениях асинхронной программы на каждом этапе исполнения будут инициированы, вообще говоря, разные подмножества А-блоков. Это обстоятельство (недетерминизм исполнения А-программы) сильно усложняет отладку асинхронной программы.

А-программа считается завершенной, когда ни один блок не выполняется и ни один А-блок не может быть инициирован, так как для всех κ=1,...,*m* значение *tr(ak)*=**false** .

Выполнение А-блока A_k состоит в исполнении:

- процедуры *ak* с теми значениями, которые имеют переменные из in(*ak*) в текущий момент, при этом вычисляются значения переменных из out(*ak*),
- управляющего оператора *c(ak).*

Следовательно, спусковые функции и управляющие операторы - это те средства, с помощью которых задается управление в асинхронных программах .

П1

integer *x, y, z*;

input (x,y,z)

asynchronous_do

tr(ak)	*ak*
x<y V *x<z*	*x:=x* + 1;
y<z V *y<x*	*y:=y* + 1;
z<x V *z<y*	*z:=z* + 1;

end_do

Программа примера **П1** позволяет в некотором порядке уравнять значения переменных *x, y, z*, доведя их с помощью прибавления единицы до значения наибольшей их них (рис. 4.1). Это программа максимальной непроцедурности, в ней нет ограничений, которые бы не были обусловлены наличием информационной зависимости между операторами.

П2.Большая непроцедурность часто мешает эффективно выполнить программу и в таких случаях требуется уменьшать непроцедурность программы (представления алгоритма), сделать управление более жестким, более определенным, не оставляющим места на размышления в ходе вычислений. Для программы примера **П1** это можно сделать, убрав дизъюнктивные члены из спусковых функций, что усилит управление (увеличит множество ограничений в нем).

integer *x, y, z*;

input (x,y,z)

asynchronous_do

tr(ak)	*ak*
x<y	*x:=x* + 1;
y<z	*y:=y* + 1;
z<x	*z:=z* + 1;

end_do

П3.Программа нижеследующего примера отличается от программы примера **П2** тем, что в спусковые функции добавлены новые конъюнктивные и дизъюнктивные члены. Они добавляют новые ограничения в управление и в результате получается почти последовательная программа.

Параллельное выполнение двух А-блоков возможно лишь при условии равенства значений двух переменных (при равенстве значений всех трех переменных программа завершается).

input (x,y,z)

asynchronous_do

tr(ak)	*ak*
x<z & x<*y* V(x=z&x<y)V(x=y&x<z)	*x*:=*x* + 1;
y<*x* & *y*<*z* V(y=x&y<z)V(y=z&y<x)	*y*:=*y* + 1;
z<y & z<*x* V(z=y&z<x)V(z=x&z<y)	*z*:=*z* + 1

end_do

П4.В примерах **П1** - **П3** А-блоки не имели управляющего оператора (более точно, был пустой управляющий оператор). Рассмотрим теперь организацию конвейерного исполнения А-блоков. Схема программы показана на рис. 4.2.

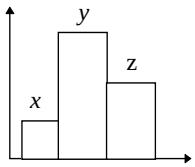


Рис. 4.1.

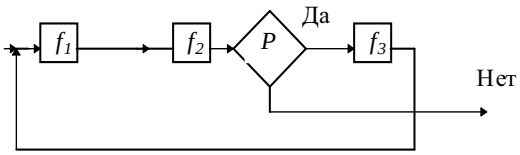


Рис. 4.2.

Асинхронная программа, реализующая этот конвейер (в программе не показаны переменные, обрабатываемые этой программой, демонстрируется только конструирование управления), может иметь вид.

integer *x, y, z*;

asynchronous_do

x:=1; *y*:=*z*:=0;

tr(ak)	*ak*	*c(a_k)*
x=1	*f1*	**[***y*:=1; *x*:=0**]**;
y=1	*f2*	**[***z*:=1, *y*:=0**]**;
P & *z*=1	*f3*	**[***x*:=1; z:=0**]**;

end_do

В этой программе операторы присваивания *x*:=1; *y*:=*z*:=0; значений управляющим переменным разрешают инициировать операцию *f1* и запрещают инициирование операций *f2* и *f3*. Таким образом, в начальный момент времени сможет быть инициирован только А-блок *f1*. После выполнения операции *f1* управляющий оператор *c1* (здесь это оператор **[***y*:=1; *x*:=0**]**) запретит исполнение А-блока *f1* и разрешит исполнение А-блока *f2* и т.д. Цикл конвейера завершится, когда предикат *P* станет ложным и не позволит инициировать А-блок *f3*.

Асинхронное программирование оказалось весьма трудоемким делом, особенно отладка программы, в силу обилия возможностей совершить ошибки в синхронизации процессов. Рассмотрим некоторые проблемы асинхронного программирования.

4.1.2.Некорректное вычисление данных

Возникновение некорректных данных иллюстрирует следующий простой пример. Пусть необходимо начислить зарплату и вычислить сумму денег, подлежащих выдаче на руки. Оставляя в стороне излишние здесь детали, будем предполагать, что зарплату составляет некоторая базисная зарплата N_0 плюс надбавки N_1, N_2, ... , N_n (выражаются в процентах к базисной зарплате) минус налог N_{n+1} (выражаются в процентах к начисленной сумме).

Пусть процессы P_0, P_1, P_2, ... , P_n, P_{n+1} соответственно выполняют эти операции. Понятно, что для вычисления корректного результата процесс P_{n+1}, вычисляющий сумму налога, должен выполняться последним, процесс P_0 - первым, а процессы P_1, P_2, ... , P_n могут исполняться в любом порядке, в том числе и параллельно, хотя само суммирование должно, конечно, выполняться последовательно. Все другие варианты выполнения процессов приведут к некорректным результатам.

Такого сорта ошибки асинхронных программ крайне неприятны, трудно локализуемы и неповторимы. Если позволить процессам P_1, P_2, ... , P_n, P_{n+1} выполняться асинхронно, то при разных выполнениях асинхронной программы могут получаться разные результаты и повторить предшествующее тестирование удастся только случайным образом. Понятно, что $(P_0 + P_1 + P_{n+1} + P_2 + ... + P_n) \neq (P_0 + P_{n+1} + P_1 + P_2 + ... + P_n)$, здесь имеется ввиду, что процессы выполняются в написанном порядке.

4.1.3.Некорректное считывание данных

Следующий пример иллюстрирует этот тип ошибки. Пусть в банке *A* есть счет *acc1*, на котором находится 500 тыс. руб., а в банке *B* - счет *acc2*, на котором находится 300 тыс.руб, и необходимо переслать 100 тыс.руб. со счета *acc1* на счет *acc2*. Сумма денег на обоих счетах неизменна до и после выполнения пересылки и равна 800 тыс. руб. Пусть процесс P_1 посылает деньги из банка *A* в банк *B*, а процесс P_2 принимает посланные деньги в банке *B*. Процессы схематически могут быть описаны так:

Первоначально

A.acc1=500 тыс.руб.

B.acc2=300 тыс.руб.

Процесс P_1	**Процесс** P_2
A.acc1:=*A.acc1* - 100;	**receive** (*x*,*A*,*y*);
x := 100	
send (*x*,*B*,*y*);	*B.acc2* := *B.acc2* + y;

Результат

A.acc1=400 тыс.руб.

B.acc2=400 тыс.руб.

В процессе P_1 счет *A.acc1* вначале уменьшается на 100 тыс. руб., а затем 100 тыс. руб. посылаются в банк *B*. В процессе P_2 сначала принимаются 100 тыс. руб. из банка *A*, а затем увеличивается сумма на счету *B.acc2*.

Однако процесс *P*, контролирующий перевод денег и проверяющий сохранение значения суммы *A.acc1+B.acc2*, может считывать ошибочные данные. Понятно, что если *P* будет считывать данные строго до или после выполнения операции перевода денег, то результат суммирования будет одинаков. Однако в ходе выполнения процессов P_1 и P_2 при разных вариантах считывания значений *A.acc1* и *B.acc2* сумма *A.acc1+B.acc2* может равняться 700 тыс.руб. (если значение *A.acc1* было считано после его изменения, а значение *B.acc2* - до изменения) , 800 тыс.руб. и 900 тыс.руб. при других возможных вариантах считывания данных.

4.2.Message passing interface

Широко известным примером реализации модели асинхронного программирования является message passing interface (MPI). Этот метод программирования определяет программу как систему взаимодействующих процессов и стандартизует обмен данными. Обмен всегда происходит через каналы, связывающие два процесса. Такой канал реализуется очередью сообщений, каждый процесс сам определяет свою готовность начать исполнение или завершить его.

В разных формах MPI изучался в асинхронных вычислениях с конца 60-х годов как у нас в стране[1], так и за рубежом. В настоящее время MPI используется практически во всех коммерческих мультикомпьютерах в качестве основного средства параллельного программирования. Идеи программирования с MPI рассматриваются в самой общей форме.

4.2.1.Определение MPI

Предполагается, что программа определяется как система независимых, параллельно протекающих взаимодействующих процессов. Как обычно, здесь *процесс* - это исполняющаяся программа со всеми обрабатываемыми данными. Исполнение этой же программы с другими данными определяет другой процесс.

Взаимодействие определяется как посылка сообщения из одного процесса в другой и прием сообщения. *Сообщение* - любая последовательность битов, которая чаще структурируется и определяется как значение набора (кортежа, структуры) переменных. Сообщение посылается в *канал* в одном процессе и выбирается из одноименного канала в другом процессе.

Канал может рассматриваться как очередь. Оператор **send** записывает сообщение в канал, в канале может накопиться очередь не выбранных сообщений. Оператор **receive**

[1] Например, был реализован в начале 80-х годов в языке ПОЛЯР в форме асинхронного канала.

выбирает сообщение из канала, если оно есть в канале. Если очередь канала не ограничена, мы говорим об *асинхронном* MPI.

В программе для использования MPI прежде всего следует описать каналы, например:

ch *queue* (<*описание_переменной*>)

Описание определяет канал *queue*, который в состоянии хранить значения переменной канала описанного типа. Переменная канала может быть простой или структурированной. Возможны описания вида:

ch *symbol* (**char**);

ch *data* (*day*, *month*, *year* : **integer**);

ch *personal_date* (*name* : **string**, *age* : **integer**);

Запись сообщения в канал производится оператором **send**.

send <*имя_канала*> (<*выражение*>);

Выражение должно вырабатывать значение переменной, описанной в определении канала. Например:

send *data* (*d+1*, *m*, *y*);

Переменные *d*, *m* и *y* определяются в процессе-отправителе и должны содержать целые значения, как это и определено в описании канала *data*.

Данные выбираются из канала оператором **receive**.

receive *personal_date* (*n*, *a*);

Переменные *n* и *a* определяются в процессе-получателе и должны быть описаны так же, как переменная канала. После выполнения оператора **receive** переменная *n* может содержать значение “Иванов И.И.”, а переменная *a* - значение “30”.

Если канал предполагается неограниченным (асинхронный MPI), то оператор **send** может быть выполнен неограниченно много раз (*не блокируемый оператор*).

Оператор **receive** получает значение из канала, если он не пуст. Если в канале нет значения, выполнение оператора **receive** задерживается (*блокируемый оператор*) до появления значения в канале.

Каналы описываются как глобальные объекты, поскольку они разделяются процессами программы. Можно определить массив каналов, например:

ch *data* [1:k](*day*, *month*, *year* : **integer**);

Допускается, чтобы несколько процессов посылали сообщения в один и тот же канал. Аналогично, несколько процессов могут получать данные из одного канала. Описанный канал в начальный момент пуст.

П1.Одно из типичных межпроцессных взаимодействий определяет взаимодействия клиента и производителя. Производитель создает продукт и продает его многим клиентам. Программа их взаимодействий может иметь вид.

ch *request* (**integer**, <описание_запроса_клиента>);

ch *reply* (**integer**, <описание_результата>);

% <описание_запроса_клиента> - это описание типов переменных, значения которых специфицируют запрос клиента к производителю. Значение переменной, описанной как **integer**, определяет уникальный номер клиента.

<описание_результата> - описание переменных, специфицирующих ответ производителя на запрос клиента %

Producer :: **var** *index, ...*;

do *true* → **receive** *request* (*index, ...*); % ожидание запроса клиента%

..... % обработка запроса клиента и формирование ответа%

send *reply* [*index*] (...); % ответ клиенту %

od;

Client [*i* : 1...*n*] :: **send** *request* (*i, ...*); % обращение к *Producer* %

receive *reply* [*i*] (...); % ожидание ответа *Producer* %

В программе определены *n*+1 независимых параллельно протекающих процессов: один процесс-производитель *Producer* и *n* процессов-клиентов *Client*. Именно так могут быть описаны взаимодействия файл-сервера и программ, запрашивающих (открывающих) нужные им файлы. При этом файл-сервер ответственен за то, чтобы только одна программа получала доступ к файлу по записи. Это же управление годится для программирования удаленного доступа к дисковому файлу нескольких программ в сети. Много подобных задач в разных вариантах возникает при организации обработки данных в сети.

4.2.2.Параллельная программа разделения множеств

Определения MPI очевидны и просты. Так же просты и очевидны средства программирования в MPI. К сожалению, их простота и очевидность только кажущиеся.

В качестве примера рассмотрим программу разделения множеств, ее частичная корректность была доказана, однако лишь недавно[2] было формально доказано отсутствие свойства тотальной корректности[3] программы.

И это при том, что программа совершенно тривиальна, в ней попросту не на что смотреть! Это хороший пример, показывающий, что простое тестирование, без формального обоснования правильности программы, не в состоянии обеспечить правильность программы.

[2] Ю.Г. Карпов. Анализ корректности параллельной программы разделения множеств. Программирование, № 5, 1996

[3] Программа называется *корректной*, если при остановке она вырабатывает правильный результат. Программа называется *тотально корректной*, если она всегда останавливается и всегда вырабатывает

Следует также обратить внимание на опасную кажущуюся правильность частично корректных, но не тотально корректных программ. Нередко такая программа долгое время нормально работает и обнаруживает ошибку в самый неподходящий момент.

Пусть заданы два множества натуральных чисел *S* и *T*. Сохраняя мощность множеств *S* и *T* необходимо собрать в *S* наименьшие элементы множества $S \cup T$, а в *T* - наибольшие.

Последовательный алгоритм и программа очевидны: множества *S* и *T* сливаются, затем слитое множество упорядочивается и вновь разделяются на на множества *S´* и *T´*, удовлетворяющие условиям задачи.

Для параллельного асинхронного решения задачи используется следующий алгоритм.

-1.Определяются два параллельно протекающих процесса *Small* и *Large*.

-2.Процесс *Small* выбирает максимальный элемент в множестве *S*, а процесс *Large* параллельно (в то же самое время) находит минимальный элемент во множестве *T*.

-3.Процессы *Small* и *Large* синхронизуются и обмениваются данными: наибольшее значения множества *S* пересылаются процессом *Small* процессу *Large* для включения в множество *T*, а наименьшее значения множества *T* пересылаются процессом *Large* процессу *Small* для включения в множество *S*.

-4.Далее циклически повторяются шаги 3 и 4.

-5.Программа останавливается, когда наибольший элемент в множестве *S* окажется меньше наименьшего элемента в множестве *T*.

По завершении программы все элементы множества *S* должны оказаться не больше любого элемента множества *T*, а мощности этих множеств не изменяются.

Программа состоит из двух параллельных процессов, *P=[Small||Large]*.

P=[Small||Large]:

Small::	*Large***::**
mx:=max(S); α**!** *mx;* *S:= S -{mx};*	α**?***y;* *T:=T∪{y};* *mn:=min(T);*
β**?** *x;* *S:= S ∪{x};* *mx:=max(S);*	β**!** *mn;* *T:=T-{mn};* *mn:=min(T);*
**[mx > x →*	**[mn < y →*
α**!** *mx;* *S:=S-{mx};*	*α**?** y; T:=T∪{y}; mn:=min(T);*
β**?** *x; S:=S∪{x}; mx:=max(S);*	β**!** *mn;* *T:=T-{mn}; mn:=min(T)*
] **stop**	] **stop**

Программу можно прокомментировать следующим образом. Определены два процесса *Small* и *Large*. Символ || разрешает параллельное исполнение процессов *Small* и *Large*, оператор * задает циклическое исполнение (итерацию), пока истинно условие цикла.

Процессы связаны однонаправленными каналами α и β. По каналу α процесс *Small*

правильный результат. Корректные, но не тотально корректные программы исключительно опасны. Они

передает данные в процесс *Large*, а данные из процесса *Large* передаются в процесс *Small* по каналу β.

Оператор **!** задает передачу данных (аналог оператора **send**), а оператор **?** - их прием (аналог оператора **receive**). В частности, оператор α**!** mx в процессе *Small* задает передачу значения переменной *mx* в канал α**,** а оператор α**?**y в процессе *Large* определяет прием значения из канала α и присваивание этого значения переменной y.

В программе *[Small||Large]* одновременное выполнение оператора α!mx в процессе *Small* и оператора α?y в процессе *Large* (их выполнение синхронизуется, т.е., выполнение одного из них в одном из процессов задерживается до тех пор, пока другой оператор не начнет выполняться в другом процессе) имеет семантику "удаленного присваивания" y:=mx. Аналогична семантика взаимодействия по каналу β.

Обозначим S^0 и T^0 – начальные множества, а S^{Term} и T^{Term} – заключительные их значения. При правильном завершении программы ожидается выполнение соотношений (в соответствии с начальными условиями задачи):

(C1). Объединение множеств не изменилось: $S^{Term} \cup T^{Term} = S^0 \cup T^0$;

(C2). Мощности множеств сохранились: $|S^{Term}| = |S^0|$, $|T^{Term}| = |T^0|$;

(C3). Каждый элемент S^{Term} не больше любого элемента T^{Term} : $max(S^{Term}) \le min(T^{Term})$.

Частичная корректность этой программы состоит в том, что если множества S^0 и T^0 конечны и непусты, то после нормального *завершения* программы (т.е. когда каждый из процессов выходит на свой **stop**) свойства (C1), (C2) и (C3) выполняются. Тотальная корректность ее состоит в том, что если множества S^0 и T^0 конечны и не пусты, то программа завершается правильно и свойства (C1), (C2) и (C3) непременно выполняются после этого завершения.

Оставляя в стороне формальные детали, весьма поучительно рассмотреть технологические приемы, приводящие к пониманию того, что программа не является тотально корректной.

4.2.3.Коммуникационно-замкнутые слои параллельной программы

Это понятие вводится для упрощения *верификаци*и (доказательства правильности) параллельных программ. Основная идея здесь - это разбиение каждого процесса P_i параллельной программы $P::=[P_1|| \ ...||P_n]$ на последовательность его операторов: $Pi=Q_{i,1};Q_{i,2};...Q_{i,k}$ (*k* может быть выбрано одним и тем же для всех процессов, если допустить возможность использовать в качестве $Q_{i,r}$ пустой оператор). Таким образом, параллельная программа *P* может быть представлена как:

$$P=[(Q_{1,1}; Q_{1,2};...; Q_{1,k})||...||(Q_{i,1};Q_{i,2};...;Q_{i,k})|| \ ... \ ||(Q_{n,1};Q_{n,2};...;Q_{n,k})].$$

создают видимость правильной работы и, как правило, отказываются работать в самый нужный момент.

Эту программу можно изобразить матрицей (рис. 4.3)

P_1	$Q_{1,1}$	$Q_{1,2}$	...	$Q_{1,j}$	... $Q_{1,k}$
	...				
P_i	$Q_{i,1}$	$Q_{i,2}$	...	$Q_{i,j}$	... $Q_{i,k}$
	...				
P_n	$Q_{n,1}$	$Q_{n,2}$	...	$Q_{n,j}$	... $Q_{n,k}$

Рис. 4.3

Каждая i-я строка изображает процесс P_i как последовательность операторов: $Pi=Q_{i,1};Q_{i,2};...Q_{i,k}$.

Параллельная программа $L_j=[Q_{1,j}||Q_{2,j}||...||Q_{n,j}]$ называется j-м слоем параллельной программы P (j-й столбец матрицы).

Слой L_j называется *коммуникационно-замкнутым*, если при всех вычислениях P взаимодействие процессов $P_1||$ $...||P_n$ происходит только внутри этого слоя, или, иными словами, ни одна команда взаимодействия среди операторов $Q_{r,j}$ при всех выполнениях P не будет синхронизироваться (*сочетаться*) с командами взаимодействия из операторов $Q_{t,i}$ при $i \neq j$.

Тогда последовательность слоев $L_1;...;L_k$ представляет собой параллельную программу:

$$P^* = L_1;...;L_k = [Q_{1,1}||Q_{2,1}||...||Q_{n,1}];... ;[Q_{1,k}||Q_{2,k}||...||Q_{n,k}],$$

В программе P^* все L_j исполняются последовательно в порядке перечисления, а операторы каждой L_j исполняются параллельно. Программа P^* называется *безопасной*, если и только если все ее слои коммуникационно-замкнуты.

Если программа P^* безопасна, то вместо верификацию всей параллельной программы P можно проводить ее послойную верификацию, т.е., доказывать утверждение $\{p_0\}L_1\{p_1\},...,\{p_{k-1}\}L_k\{p_k\}$ вместо утверждения $\{p_0\}P\{p_k\}$. Здесь, как обычно, $\{s\}P\{q\}$ обозначает утверждение, что программа P частично корректна по отношению к предусловию s и постусловию q (вход-выходные соотношения), при этом, если до начала исполнения программы P предикат s истинен, то после исполнения P предикат q тоже истинен.

Таким образом, если параллельную программу P удается разбить на последовательность коммуникационно-замкнутых слоев, то доказательство ее (частичной) корректности сводится к последовательному доказательству вход-выходных соотношений для каждого слоя. Это существенно упрощает анализ корректности параллельной программы.

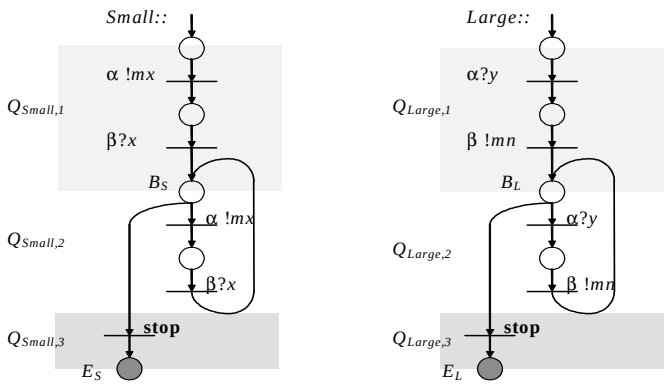


Рис. 4.4.

Процессы *Small* и *Large* разбиваются на коммуникационно-замкнутые слои совершенно естественно. Разбиение приведено на рис. 4.4.

На рисунке показаны только “синхронизационные скелеты” параллельных процессов. В первый слой входят операторы до цикла, во второй слой - операторы внутри цикла, третий слой составляет оператор **stop** после выхода из цикла. Процесс правильно завершается, если он заканчивает вычисления в заключительном состоянии: процесс *Small* в состоянии E_S , а процесс *Large* в состоянии E_L. В общем случае процессы могут:

а) оба завершиться нормально,

б) оба не завершатся,

в)один завершится, а другой нет, например, при невозможности

выполнить операцию синхронного взаимодействия.

Условия B_S и B_L определяют условия окончания циклов в процессах; они равны соответственно $mx \le x$ и $mn \ge y$.

4.2.4.Когерентность параллельных программ

Для упрощения верификации параллельной программы уже при ее проектировании на нее можно наложить дополнительные требования, облегчающие ее понимание и анализ, и заранее устраняющие некоторые типы возможных ошибок. Одним из таких требований является *когерентность* параллельной программы.

Неформально, требование когерентности означает:

-каждый раз, когда процесс хочет послать сообщение другому процессу, его партнер готов принять это сообщение, т.е., обязательно выходит на оператор приема сообщения;

-каждый раз тогда, когда процесс хочет получить сообщение некоторого типа от другого процесса, его партнер посылает ему это сообщение.

В когерентной программе невозможна ситуация, когда в одном процессе выполняется оператор α**!** mx, а процесс-партнер не может выйти на исполнение оператора α**?**y.

Если параллельная программа $P::=[P_1||...||P_n]$ разбита на коммуникационно-замкнутые слои $Pi= Q_{i,1};Q_{i,2};...Q_{i,k}$,, то требование когерентности состоит не только в том, что при всех вычислениях P ни одна команда взаимодействия среди операторов $Q_{r,j}$ при всех выполнениях P

не будет синхронизироваться (сочетаться) с командами взаимодействия из операторов $Q_{t,i}$ при $i \neq j$, но и в том, что каждая такая команда взаимодействия *обязательно будет сочетаться* с некоторой командой взаимодействия из операторов того же самого слоя. Для параллельной программы *P=[Small || Large]* такая синхронизация показана на рис. 4.5.

Очевидно, что требование когерентности для этой программы выполняется тогда и только тогда, когда условия прекращения циклов в программах Small и Large тождественны при всех прохождениях циклов.

Некогерентность, с другой стороны, ведет к блокировке этой программы, т.е., к возникновению ситуации, когда один из процессов выходит из цикла и переходит в заключительное состояние, а другой не выходит из своего цикла и "зависает" на операции синхронного взаимодействия внутри цикла, бесконечно ожидая партнера.

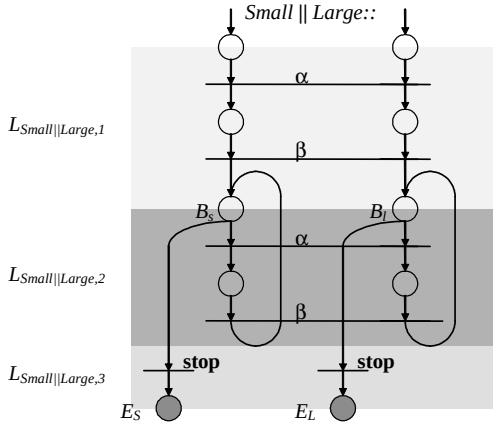


Рис. 4.5.

Таким образом, условие $B_S \equiv B_L \equiv$ И, или, что то же, $mx \leq x \equiv mn \geq y$ при *каждой* проверке условий циклов в обоих программах, является необходимым условием тотальной корректности этой параллельной программы.

4.2.5.Анализ программы разделения множеств

Для анализа программы используем "истории" взаимодействий. Вводятся вспомогательные переменные, которые хранят истории взаимодействия по каждому каналу программы.

Историческая переменная - это просто массив значений, последовательно переданных по соответствующему каналу. Пусть h_α и h_β такие исторические переменные для каналов α и β соответственно, тогда компонент $h_\alpha[k]$ содержит k-е значение, посланное по каналу α при выполнении операции $\alpha!e$.

Проведем анализ первого слоя :

$Q_{Small,1}$::	$Q_{Large,1}$::
mx:=max(S); α! *mx; S:=S-{mx};*	α?*y; T:=T∪{y}; mn:=min(T);*
β? *x; S:=S∪{x}; mx:=max(S)*	β! *mn; T:=T-{mn}; mn:=min(T)*

Для того, чтобы процессы $Q_{Small,1}$ и $Q_{Large,1}$ завершились, необходимо и достаточно, чтобы множество S содержало хотя бы один элемент, т.е. $|S|>0$. По завершении каждого из этих параллельных процессов первого слоя будут справедливы следующие соотношения:

для $\boldsymbol{Q_{Small,1}}$:

$mx^0 = max(S^0);$

$h_\alpha[0] = mx^0;$

$x^1 = h_\beta[0];$

$S^1=(S^0-\{max(S^0)\})\cup\{x^1\};$

$mx^1 = max(S^1);$

для $\boldsymbol{Q_{Large,1}}$:

$y^1 = h_\alpha[0];$

$mn^0 = min(T^0 \cup \{ y^1 \});$

$h_\beta[0] = mn^0;$

$T^1=T^0\cup\{y^1\}-\{min(T^0\cup\{y^1\})\};$

$mn^1 = min(T^1);$

Эти соотношения просто описывают, что было сделано при исполнении операторов первого слоя.

Рассмотрим теперь второй слой:

$\boldsymbol{Q_{Small,2}}$::

*[$mx > x$ →

α! mx; $S:=S-\{mx\};$

β? x; $S:=S\cup\{x\}$; $mx:=max(S);$

]

$\boldsymbol{Q_{Large,2}}$::

*[$mn < y$ →

α? y; $T:=T\cup\{y\}$; $mn:=min(T);$

β! mn; $T:=T-\{mn\}$; $mn:=min(T)$

]

Перед i-м выполнением каждого цикла для процессов $Q_{Small,2}$ и $Q_{Large,2}$ истинны следующие инварианты, что можно проверить непосредственно (где a^i -значение переменной a перед i-ым выполнением цикла):

для $\boldsymbol{Q_{Small,2}}$:

$\boldsymbol{I_{Small,2}} \equiv h_\alpha[i-1] = mx^{i-1} \wedge$

$x^i = h_\beta[i-1] \wedge$

$S^i=(S^{i-1}-\{max(S^{i-1})\})\cup\{x^i\}\wedge$

$mx^i = max(S^i);$

для $\boldsymbol{Q_{Large,2}}$:

$\boldsymbol{I_{Large,2}} \equiv y^i = h_\alpha[i-1] \wedge$

$h_\beta[i-1] = min(T^{i-1} \cup \{ y^i\})\wedge$

$T^i=T^{i-1}\cup\{y^{i-1}\}-\{min(T^{i-1}\cup\{y^i\})\}\wedge$

$mn^i = min(T^i);$

Как уже говорилось, требование когерентности в этой программе соответствует требованию общезначимости формулы $mx^i \le x^i \equiv mn^i \ge y^i$. При некоторых значениях исходных множеств S и T она нарушается. Возможны два случая некогерентности:

а) $mx^i \le x^i$; $mn^i < y^i$; или, что то же: $max(S^i) \le min(T^{i-1} \cup \{ max(S^{i-1})\})$; $min(T^i) < max(S^{i-1})$;

при этом процесс *Small* завершается, а процесс *Large* продолжает выполнять цикл, что приводит к его блокировке;

б) $mx^i > x^i$ или, что то же: $max(S^i) > min(T^{i-1} \cup \{ max(S^{i-1})\});$

$mn^i \geq y^i$ $min(T^i) \geq max(S^{i-1});$

при этом процесс *Large* завершается, а процесс *Small* продолжает выполнять цикл и блокируется, бесконечно ожидая взаимодействия с процессом *Large*.

Учитывая, что $min(T^i) \geq min(T^{i-1})$ и $max(S^i) \leq max(S^{i-1})$, условия некорректного поведения параллельной программы разделения множеств можно записать проще:

а) $max(S^i) \leq min(T^{i-1});$ б) $max(S^i) > min(T^{i-1});$

$min(T^i) < max(S^{i-1});$ $min(T^i) \geq max(S^{i-1}).$

Поясним полученный результат. Исследуемая параллельная программа разделения множеств имеет целью собрать все минимальные элементы объединения двух множеств *S* и *T* в множестве *S*, а максимальные элементы $S \cup T$ - в множестве *T*, причем мощности множеств не должны измениться. Упорядочим элементы исходных множеств: множества *S* по убыванию, а множества *T* по возрастанию. На рис. 4.6а. показано, что процесс *Small*, работая на множестве *S*, пересылает его максимальные элементы в множество *T*, а процесс *Large*, работая на множестве *T*, пересылает его минимальные элементы в множество *S*.

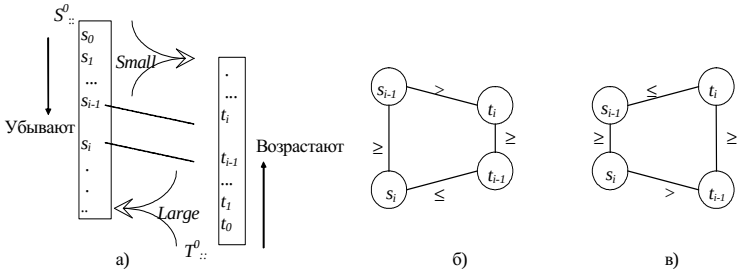


Рис. 4.6.

Полученные выше условия некорректного поведения программы определены для (*i*-1)-го и *i*-го максимальных значений множества *S* и для (*i*-1)-го и *i*-го минимальных значений множества *T*, (i =1,2, ...). Эти условия представлены диаграммами на рис. 4.6.б) и 4.6.в).

Иными словами, если между упорядоченными по убыванию элементами множества *S* и упорядоченными по возрастанию элементами множества *T* на одном и том же расстоянии от начала выполнится одно из отношений рис. 4.6,б) или рис. 4.6,в), то исследуемая программа будет работать некорректно: она входит в дедлок.

Можно указать тестовый пример, на котором эта программа работает некорректно: S={5,10,15,20}, T={17,18,30,40,60}. Этот пример относится к первому типу некорректностей при *i*=1: первое же вхождение процесса *Large* в цикл приводит к дедлоку, поскольку *Small* завершится, не входя в цикл. Однако, полагаться на возможность обнаружения этой некорректности с помощью тестов нельзя. Добавление, например, к множеству *T* любого числа элементов, меньших 15, нарушит это соотношение и программа будет работать корректно.

Ручной прокруткой можно проверить, что на множествах S={5,10,15,20}, T={14,17,18,30,40,60} программа работает правильно: сортирует множества и завершается после однократного прохождения циклов в обоих процессах.

5. ОРГАНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ В КРУПНОБЛОЧНЫХ ИЕРАРХИЧЕСКИХ МУЛЬТИКОМПЬЮТЕРАХ

5.1. Введение

С развитием модульного подхода в конструированию мультикомпьютеров к настоящему времени появилась возможность формировать мультикомпьютеры разнообразной структуры из серийных блоков вычислительной техники. Такие мультикомпьютеры состоят обычно из управляющей универсальной ЭВМ и нескольких вычислительных узлов. Каждый вычислительный узел мультикомпьютера – это обычный компьютер, состоящий из процессора, памяти и коммутационных каналов межузловой связи. Такой узел называется *процессорным элементом* - ПЭ.

Наибольший интерес для быстрого и широкого использования имеют *крупноблочные* мультикомпьютеры, которые собираются на базе крупных устройств. Типичными представителями таких устройств являются универсальные и специализированные микропроцессоры (они используются в качестве процессоров в узлах мультикомпьютеров), каждый из которых имеет собственное устройство управления, регистры, кэш-память, в них можно переслать данные и программы для исполнения. Из таких компонентов и собираются

мультикомпьютеры, у которых каждый узел является самостоятельным независимо работающим компьютером. Коммерческие мультикомпьютеры делают масштабируемыми (scalable), допуская в широком диапазоне изменения числа ПЭ и объема их памяти в различных установках.

Можно указать на следующие преимущества крупноблочных мультикомпьютеров.

1).Хорошее значение соотношения производительность/стоимость. Высокая производительность на узком классе задач может достигаться за счет специализации ПЭ и за относительно невысокую цену, а высокая степень универсальности мультикомпьютера обеспечивается включением в его состав ПЭ различного типа;

2).Для каждой предметной области и даже для решения конкретной большой задачи могут быть созданы проблемно-ориентированные мультикомпьютеры, содержащие только ПЭ нужного типа и в требуемом количестве, что удешевляет стоимость всего мультикомпьютера;

3).Быстрое создание мультикомпьютеров, так как компоненты включаются в них со своим штатным математическим обеспечением и необходимо проведение только интегрирующих доработок;

4).Сравнительная простота переноса существующего прикладного математического обеспечения на мультикомпьютер

за счет его эволюционного развития, а не коренной переделки. По этой причине наиболее эффективным использование крупноблочных мультикомпьютеров будет тогда, когда на базе существующего прикладного математического обеспечения и серийного оборудования необходимо на 1-3 порядка увеличить объем вычислений.

В мире известно сейчас много коммерческих мультикомпьютеров, собранных на базе разных микропроцессоров. Весьма практично объединять в мультикомпьютер - кластер (cluster) - несколько рабочих станций специальным коммутатором.

Создание крупноблочных мультикомпьютеров наталкивается на ряд трудностей. Основная из них состоит в том, что слишком велико оказывается многообразие различных возможных структур мультикомпьютеров и алгоритмов и далеко не всегда они согласуются друг с другом, следствием чего может быть резкое падение производительности мультикомпьютера из-за неудачного отображения алгоритма на его ресурсы. Решение проблемы эффективного отображения массового алгоритма в ресурсы мультикомпьютеров в общей постановке вряд ли возможно, и потому большой проблемой становится как создание эффективных параллельных программ, так и обеспечение их переносимости, накопление фонда прикладных программ.

Рассмотрим проблемы организации параллельных вычислений на иерархических мультикомпьютерах, которые имеют в этом классе хорошие стандартные решения в силу сравнительно низких затрат на организацию вычислений.

Для организации параллельной обработки больших объемов промышленных данных, учитывая регулярность массовых вычислений[1], в крупноблочных мультикомпьютерах может быть использован сравнительно простой подход, сформулированный в форме метода линеаризации массовых вычислений. В нем определяется класс иерархических мультикомпьютеров, ограничивающий произвол в компоновке мультикомпьютера, но не содержащий ограничений на количество компонент в составе мультикомпьютера. На структуру алгоритмов также вводятся ограничения, которые выделяют класс алгоритмов, эффективно реализуемых на иерархических мультикомпьютерах. В совокупности это обеспечивает создание систем параллельного программирования (СПП), в которых прикладные параллельные программы собираются из готовых фрагментов, они динамически эффективно настраиваются на доступные ресурсы

[1] К массовым вычислениям приводят решение различных численных задач, например, задач обработки изображения и сейсмических данных, решение систем дифференциальных уравнений разностными методами, моделирование разлета облака плазмы методом частиц и т.п.

мультикомпьютера и переносимы в выделенном классе мультикомпьютеров.

Итак, метод линеаризации массовых вычислений определяет:

-класс допустимых мультикомпьютеров - иерархические,

-допустимые структуры алгоритмов - линейные алгоритмы,

- фиксированный mapping алгоритм.

Все вместе это позволяет создать весьма эффективное и универсальное системное параллельное программное таких мультикомпьютеров. Метод был реализован (весь состав параллельного системного программного обеспечения) в Вычислительном центре СО АН СССР в ходе работ по созданию кластера «Сибирь» в 1987-1989гг.

5.2. Иерархические мультикомпьютеры

Для получения в СПП приемлемого решения проблемы отображения алгоритма на ресурсы мультикомпьютера, его структуру следует разумно ограничить. Необходимо так выбрать систему взаимно дополняющих ограничений на структуру алгоритма и на структуру мультикомпьютера, чтобы СПП получилась простой и доступной прикладному пользователю, ограничения легко удовлетворялись и множество эффективно реализуемых алгоритмов достаточно широко.

Что можно сказать о структуре крупноблочного мультикомпьютера? Прежде всего следует отметить, что не должно быть конкретных требований типа: нельзя использовать в мультикомпьютере более десяти однотипных ПЭ. Класс крупноблочных мультикомпьютеров должен определяться набором правил для построения допустимого, правильного мультикомпьютера.

Далее, структура связей компонент мультикомпьютера должна носить такой регулярный характер, который бы не изменялся от добавления в состав мультикомпьютера нового или удаления старого компонента. Например, характер связей в мультикомпьютере - объединение ПЭ общей шиной - не меняется при добавлении/удалении ПЭ. И, наконец, регулярность структуры мультикомпьютера должна быть такова, что добавление новых или удаление старых (например, отказ ПЭ) компонент не должны сделать неэффективным отображение алгоритма на ресурсы мультикомпьютера. Производительность исполнения программы на мультикомпьютере должна до некоторого предела возрастать или уменьшаться пропорционально расширению или деградации его ресурсов.

Последние два свойства структуры мультикомпьютера крайне важны. Дело в том, что нередко задача отображения алгоритма на ресурсы мультикомпьютера решается статически, до начала вычислений и все принятые решения по

распределению вычислительных ресурсов, в первую очередь ПЭ, фиксируются в тексте программы. Для крупноблочного мультикомпьютера такой подход неприемлем, т.к. мультикомпьютер может иметь много ресурсов и далеко не каждая программа все их использует и, кроме того, разные установки имеют разные состав ПЭ и структуру коммутационной сети. Следовательно, вычислительные ресурсы мультикомпьютера надо разделять динамически.

Здесь предполагается, что массовый алгоритм представляется потенциально бесконечным множеством операций, шагов вычислений. При конструировании программы множество операций должна быть отображена на ресурсы вычислительной системы, после чего может быть сформирован текст программы. Отображение (mapping) выполняется алгоритмом, который будем называть М-алгоритмом. В процедурных системах программирования М-алгоритм частично реализуется в компиляторе и в операционной системе (например, распределение регистров, оперативной памяти для буферизации данных в операциях ввода/вывода и т.п.), а частично выполняется программистом (распределение оперативной памяти для переменных в Фортране).

Для крупноблочных мультикомпьютеров конструирование М-алгоритма проводится в два этапа:

- статический, который реализуется в компиляторе и производит распределение той части ресурсов мультикомпьютера, что может быть отведена программе до начала вычислений (память, регистры), и

- динамический, который реализуется в программе и/или в операционной системе и производит захват и перераспределение ресурсов мультикомпьютера в ходе исполнения прикладной программы (процессоры, каналы).

Если структура мультикомпьютера обладает последними двумя свойствами, то динамическая часть М-алгоритма может быть зафиксирована и вкладываться в программу при компиляции. При этом параллельная программа будет переносима в таком классе мультикомпьютеров и эффективно выполнима на каждой установке класса. Здесь, конечно, предполагается, что структура выделенных параллельной программе ресурсов мультикомпьютера не меняется в ходе ее выполнения, таким образом, если выделенное программе устройство окажется неработоспособным, она должна завершиться аварийно.

Учитывая свойства, которыми должен обладать класс мультикомпьютеров, определим его в два шага следующим образом. На первом выбираются базовые компоненты, из которых может конструироваться мультикомпьютер.

Первый компонент - универсальная ЭВМ (например, рабочая станция Sun) со всей необходимой периферией, развитыми связями с внешним миром. Она будет использоваться как управляющая ЭВМ (хост ЭВМ) для управления подчиненными ресурсами мультикомпьютера.

Второй компонент - различные ПЭ, например, на базе микропроцессоров фирм Intel, IBM, AMD и т.п. Всякий ПЭ - это автономный высокопроизводительный вычислитель (возможно специализированный) с собственной оперативной памятью и устройством управления. Работа ПЭ организуется управляющей ЭВМ, которая инициирует ПЭ, передает в ПЭ программы и данные, синхронизирует вычисления, собирает результаты счета. Не запрещено использовать универсальную ЭВМ в качестве ПЭ, т.е. как подчиненный вычислитель.

Последний компонент - канал связи, обеспечивающий передачу данных между компонентами мультикомпьютера.

На втором шаге описания класса мультикомпьютеров определяются возможные соединения базовых компонент. Всего допускается два типа соединений. Радиальное соединение обеспечивает взаимодействие между управляющей ЭВМ и ПЭ по схеме старший-подчиненный, при этом допускается взаимодействие более чем с одним ПЭ. Одинаковые ПЭ образуют подсистему, возможно наличие нескольких подсистем ПЭ.

Другой вид связи - межпроцессорный интерфейс. Это может быть шина, общая память, непосредственная связь ПЭ посредством каналов, коммутационная сеть и т.п. Он позволяет с большой скоростью производить обмен данными между ПЭ внутри подсистемы однородных ПЭ, минуя управляющую ЭВМ, и обеспечивает эффективное взаимодействие между процессорами нижнего уровня. Учитывая последние два свойства класса мультикомпьютеров, всякий межпроцессорный интерфейс должен обеспечивать, как минимум, связь $ПЭ_i$ с $ПЭ_{i-1}$ и с $ПЭ_{i+1}$. Этот интерфейс будем называть линейным (рис. 5.1)

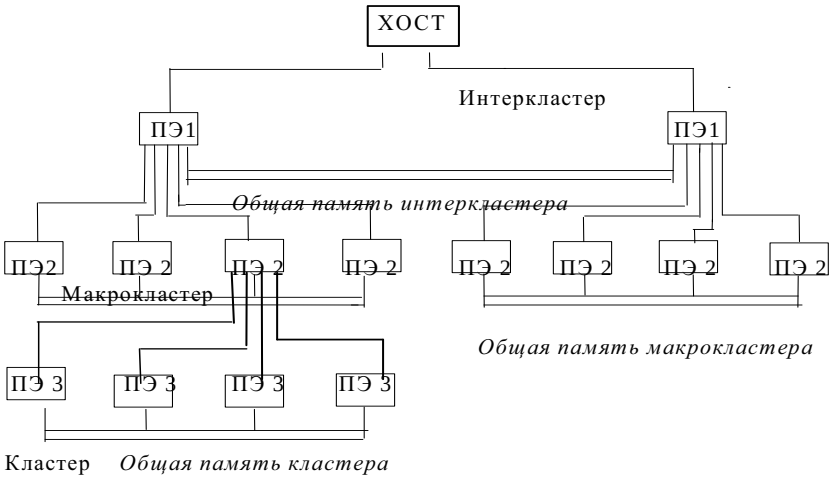


Рис. 5.1. Пример иерархического мультикомпьютера

Приведенные соединения описывают лишь общую схему взаимосвязи компонент в мультикомпьютере. Существует возможность расширения их номенклатуры, например:

подключение буферной памяти к каналам связи для организации асинхронных обменов, использование в качестве дополнительных устройств дисков и т.д. Однако такие расширения не выводят мультикомпьютер за определяемый класс, так как не изменяется схема допустимых взаимодействий между вычислителями мультикомпьютера. Комбинируя различные типы соединений, можно создавать мультикомпьютеры различной конфигурации, приспосабливая их для решения конкретного класса задач.

Каждый ПЭ в свою очередь может иметь сложную структуру, но и в этом случае ПЭ должен состоять из старшего процессора и подчиненных. Таким образом, допустимый мультикомпьютер должен иметь иерархическую структуру с линейными связями между ПЭ в пределах подсистемы однородных ПЭ. Это в свою очередь означает, что программное обеспечение мультикомпьютера должно в ходе исполнения процесса допускать порождение новых процессов, которые отправляются на исполнение в подчиненные ПЭ.

Конечно, мультикомпьютер может иметь и другие дополнительные связи между компонентами, что не ухудшит выполнение программ. На рис. 5.1 показан пример иерархического мультикомпьютера (интеркластер).

Необходимо указать на следующую характерную особенность крупноблочных мультикомпьютеров. Так как

мультикомпьютер собирается из готовых серийных крупных блоков различного назначения и применения, развивается во времени, то следует предположить, что в нём отсутствуют общие для всех компонент средства прерывания и синхронизации, вследствие чего все взаимодействия должны быть организованы программно и асинхронно.

5.3. Линейные алгоритмы

Иерархические мультикомпьютеры являются, конечно, весьма универсальными вычислительными системами и трудно представить себе практически важную комплексную задачу, которая бы не могла быть эффективно решена на них при надлежащем выборе состава ПЭ и структуры связей между ними. Теперь на пути к построению СПП для прикладного пользователя необходимо определить такие ограничения на алгоритм, чтобы в СПП можно было зафиксировать хорошо работающую динамическую часть М-алгоритма. В поиске представления алгоритма, удовлетворяющего указанным ограничениям, пользователь будет проделывать ту работу, которую должен был бы проделать компилятор, подготавливая эффективное динамическое распределение процессоров мультикомпьютера. Конечно, поиск такого представления должен быть для человека почти очевидной задачей. Условия,

определяющие необходимое представление, - условия *линейности* алгоритма - состоят в следующем:

1).На множестве операций алгоритма должен быть определен линейный порядок. В параллельной программе операции соответствует выполнение некоторой процедуры, подпрограммы, с конкретными входными данными, причем два выполнения одной и той же процедуры с разными входными данными определяют разные операции.

2).Выбранный линейный порядок должен быть таким, чтобы все взаимодействия между операциями (передачи данных от одной операции к другой) оказались линейными, то есть, если существует взаимодействие между i-ой и j-ой операциями a_i и a_j, то $j=i\pm b_k$, $k=1,2,...,n$, b_k, n - целые константы и существует такая целая константа m, что любая операция a_i взаимодействует только с операцией a_j, $a_j \in \{a_{i-m},...,a_{i+m}\}$. Такое взаимодействие называется линейным.

Множество $\{a_{i-m},...,a_{i+m}\}$ будем называть m-окрестностью операции a_i. Ограничения на структуру взаимодействий здесь не накладываются, однако предполагается, что если при отображении алгоритма на ресурсы мультикомпьютера операции m-окрестности $\{a_{i-m},...,a_{i+m}\}$ будут назначены на процессоры, то все взаимодействия операции a_i могут реализоваться при наличии необходимых линий связи, операция

a_i сможет завершиться и освободит ресурсы, которые могут быть назначены для исполнения другой операции.

Условие 2) линейности взаимодействий можно заменить в общем случае условием наличия такой целой константы алгоритма m, что любая операция a_i взаимодействует только с операцией a_j из ее m-окрестности $\{a_{i-m},...,a_{i+m}\}$. Теперь, однако, требуется, чтобы коммуникационная сеть обеспечивала любые коммуникации в m-окрестности.

Неформально, алгоритм линеаризован, если его операции линейно упорядочены и длина коммуникаций любых процессов ограничена константой. Для практики полезны алгоритмы с малой длиной коммуникаций, лучше всего, когда m=1.

Рассмотрим следующий пример. Пусть дана матрица A размерности $s \times t$ и необходимо последовательно перевычислять ее элементы (некоторая фильтрация):

$$A^0_{ij} = A_{ij},$$
$$A^1_{ij} = f_1(E_1(A^0_{ij}),b_1),$$
$$A^2_{ij} = f_2(E_2(A^1_{ij}),b_2),$$
$$\vdots$$
$$A^k_{ij} = f_k(E_k(A^{k-1}_{ij}),b_k) \qquad (1)$$

$\forall \quad i \in \{1,2,...,s\}, \quad \forall \quad j \in \{1,2,...,t\},$

k - целая положительная константа.

Здесь E_k - окружение элемента A_{ij}, b_k - матрица.

К примеру,

$$b_1 = \begin{matrix} 0 & 0,5 & 0 \\ 0.5 & 1 & 0.5, \\ 0 & 0,5 & 0 \end{matrix} \qquad E_1 = (A_{i-1,j}, A_{i+1,j}, A_{i,j}, A_{i,,j-1}, A_{i,,j+1})$$

Здесь (1) есть система рекуррентных соотношений, которые часто используются в программировании для задания массовых вычислений. В программе, реализующей алгоритм (1), функции $f_1,...,f_k$ вычисляются процедурами. Упорядочим операции алгоритма так, чтобы вычисление элементов матрицы A выполнялось бы построчно (верхний индекс функции указывает номер операции в упорядочивании):

$A^1_{22} = f^1_1(E_1(A^0_{22})b_1), \quad A^1_{23} = f^2_1(E_1(A^0_{23})b_1), \quad A^1_{24} = f^3_1(E_1(A^0_{24})b_1),$

$... , A^1_{33} = f_2^{(s-2)(t-2)}(E_2(A^1_{33})b_2), ... ,$

Ясно, что m зависит от размеров матрицы A и не является константой алгоритма. Следовательно, алгоритм в

выбранном упорядочивании не линеен. Если зафиксировать число столбцов матрицы A, то алгоритм станет линейным, хотя и с большим значением m для больших задач.

Однако этот алгоритм, как и многие другие массовые алгоритмы с регулярной структурой взаимодействий, может быть линеаризован, преобразован в другой, функционально эквивалентный алгоритм. Линеаризация алгоритма достигается грануляцией вычислений, объединением фрагментов регулярных вычислений в новую операцию с большим объемом вычислений.

Для алгоритма фильтрации линеаризация делается следующим образом. Матрица A делится на перекрывающиеся полосы $A_1,A_2,...,A_n$ (рис. 5.2). Все вычисления $A^p_{ij} = f_p(E_p(A_{ij}^{p-1}))$, $p=1,2,...,k$, выполняющиеся в полосе A_r, $r=1,...,n$, объединяются в новую операцию a_r. Операции $a_1,a_2,...,a_n$ определяют алгоритм, функционально эквивалентный (1), операции линейно упорядочены. При достаточно большом перекрытии взаимодействий нет.

Рис. 5.2.

Если значение k велико, то и число столбцов в перекрытии будет большим. Чтобы уменьшить это число, после вычисления A^p_{ij} во всей полосе A_r, перед вычислением A_{ij}^{p+1}, нужно восстанавливать

столбцы перекрытия обменом между операциями a_{r-1}, a_r, a_{r+1}. В этом случае в перекрытиях достаточно иметь всего по два столбца, но появятся линейные взаимодействия между этими операциями. Алгоритм линеен при всех размерах матрицы A таких, что в одном ПЭ еще может быть размещена полоса A_r, значение m мало и является константой при значительных изменениях размера матрицы A.

5.4. Динамическое отображение алгоритма на ресурсы мультикомпьютера

Теперь можно зафиксировать М-алгоритм. Рассмотрим характерный для массовых вычислений пример. Пусть задан файл f, состоящий из n записей. Каждую запись необходимо считать в оперативную память управляющей ЭВМ (операция *in*)), затем переслать в ПЭ (операция *put*), там обработать без взаимодействий (операция *run*), переслать результат в управляющую ЭВМ (операция *get*)) и далее во внешнюю память (операция *out*), рис. 5.3.

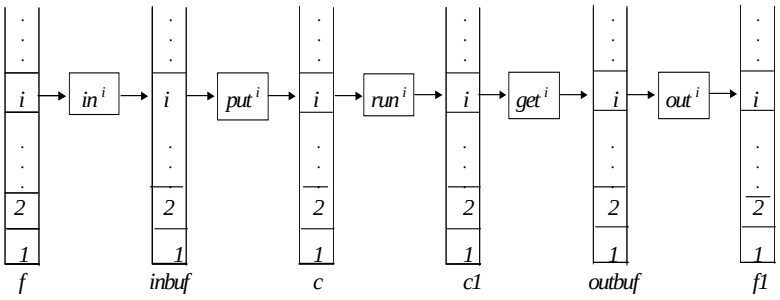


Рис. 5.3.

Алгоритм линеен. Порядок на множестве операций следующий: in^1, put^1, run^1, get^1, out^1, in^2, put^2,... . Последовательность i-х выполнений операций in^i put^i run^i get^i out^i образуют процесс p_i. Каждый процесс p_i - конечное множество последовательно исполняемых операций алгоритма. Порядок $p_1,p_2,...$ на множестве процессов линейный. Линейно упорядоченное множество процессов $\{p_1,p_2,...\}$ определяет, конечно же, тот же самый алгоритм, но с использованием более крупных операций. Знание же внутреннего устройства процессов позволяет более эффективно разделять общие ресурсы.

Динамически распределяемыми ресурсами являются ПЭ, оперативная память управляющей ЭВМ для размещения входных (inbuf) и выходных (outbuf) буферов ввода/вывода. Тогда, если доступны два $ПЭ_1$ и $ПЭ_2$, три входных буфера b_1,b_2,b_3 и два выходных буфера a_1,a_2 (рис. 5.4), то они будут приписаны

динамически процессам следующим образом *(p_1,b_1,PE1,a_1), (p_2,b_2,PE2,a_2), (p_3,b_3,PE1,a_3), (p_4,b_4,PE2,a_4), ...*

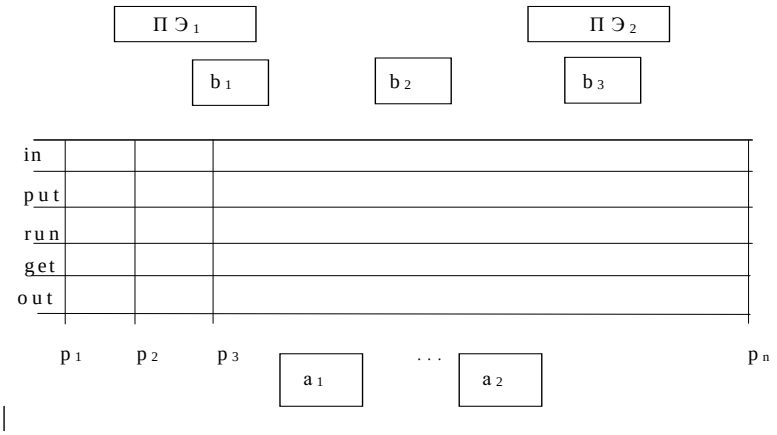


Рис. 5.4.

Такой метод распределения ресурсов называется *роллингом*. При организации вычислений процессы будут выполняться строго в порядке возрастания номеров, присвоенным им в линейном алгоритме. Процесс с меньшим номером всегда имеет преимущество при захвате очередного требуемого ресурса, более того, он не позволяет процессу с бóльшим номером “опережать” себя.

Если ПЭ - сложный вычислитель, то ему для выполнения передается конечная последовательность процессов (в заданном порядке), которые затем распределяются на подчиненные ПЭ. При работе в системе программирования, построенной на базе линеаризации массовых вычислений, программист должен знать

о роллинге и явно использовать этот факт в ходе разработки своей программы. Учитывая роллинг, он может дать СПП дополнительную информацию о структуре процессов. Для последнего примера такой информацией будут сведения о моментах захвата и освобождения ресурса. В этом случае появится возможность разделять ресурсы, одна из возможных временных диаграмм выполнения процессов приведена на рис. 5.5, многоточия обозначают время ожидания освобождения ресурсов.

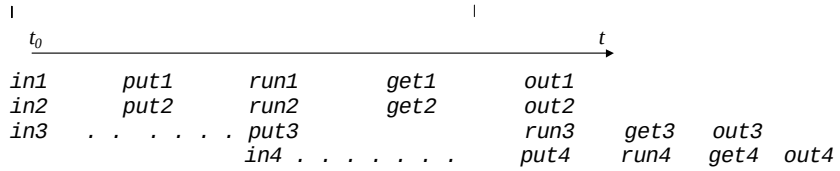


Рис. 5.5.

При отображении линейного алгоритма на иерархический мультикомпьютер линейно упорядоченное множество процессов алгоритма делится на участки, которые назначаются на ПЭ, далее каждый участок в свою очередь может быть поделен на участки и т.д.

На рис. 5.6 показана еще одна схема организации вычислений линеаризованного алгоритма умножения матриц *C=A*B*. Матрица *B* делится динамически на столько вертикальных полос, сколько ПЭ доступно в момент исполнения

программы, и вводятся в ПЭ. Матрица A делится на горизонтальные полосы $A_1,A_2,...,A_n$ так, чтобы в одном ПЭ могло быть вычислено произведение $C_{ij}=A_i*B_j$, далее полосы $A_1,A_2,...,A_n$ "составом" вводятся в $ПЭ_1$, далее сдвигаются в $ПЭ_2,...$, $ПЭ_m$, в каждом ПЭ производятся вычисления $C_{ij}=A_i*B_j$,.

При такой организации параллельных вычислений, все процессы должны выполняться примерно за одно и то же время. Ограничение трудно удовлетворить в общем случае, но для массовых вычислений это, как правило, нетрудно сделать.

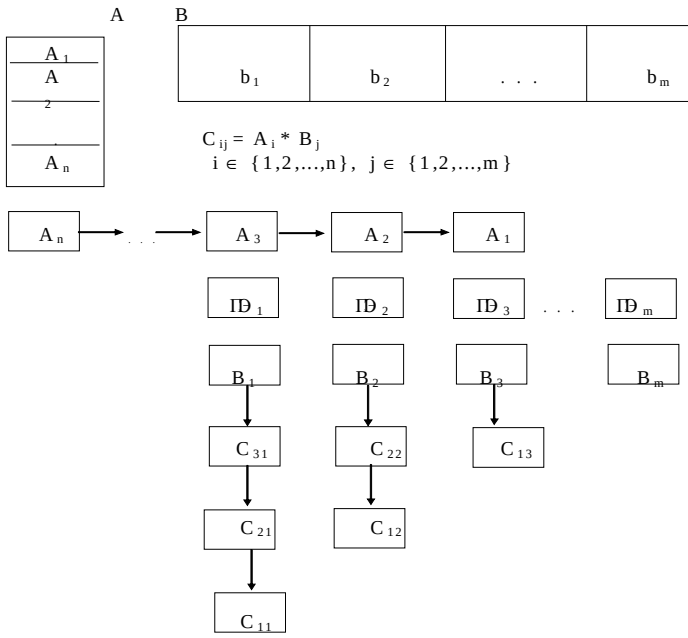


Рис. 5.6.

При реализации линеаризованного алгоритма могут возникать специфические дедлоки из-за недостатка ресурсов, пример показан на рис. 5.7. Если обмен между ПЭ идет через каналы по схеме $ПЭ_{i-1} \leftrightarrow ПЭ_i \leftrightarrow ПЭ_{i+1}$, то процессы при обмене должны быть синхронизованы, т.е., к примеру, для обмена блоком данных c^1_{12} процессы p_1 и p_2 должны дождаться друг друга, произвести передачу данных c^1_{12} (в направлении по стрелке на рисунке) и далее продолжать вычисления асинхронно.

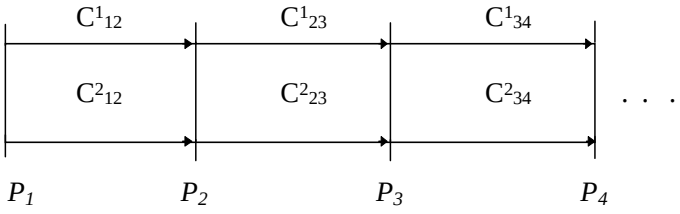


Рис. 5.7.

Если доступны только два ПЭ, то процесс p_1 будет выполняться на $ПЭ_1$, p_2 - на $ПЭ_2$, процесс p_1 сможет передать блок данных c^1_{12} процессу p_2 и продолжит вычисления. Так как процесс p_3 не выполняется (нет свободного ПЭ), то процесс p_2 не сможет передать в p_3 блок данных c^1_{23} и перейдет в состояние ожидания. По этой причине и процесс p_1 перейдет в состояние ожидания, когда понадобится передать в p_2 блок данных c^2_{12}. Если два ПЭ имеют общую память (и тогда взаимодействия

выполнятся асинхронно) или на трех ПЭ этот алгоритм будет реализован правильно. Таким образом, для каждой программы должны быть сформированы требования к минимально необходимым ресурсам мультикомпьютера, это одна из технических характеристик параллельной программы. Существует, конечно, и верхняя граница необходимых ресурсов, связанная обычно либо с отсутствием работы для ПЭ, либо невозможностью подготовить большее количество исходных данных.

Линейность алгоритма и роллинг ресурсов позволяют организовать взаимодействия и в том обычном для крупноблочных мультикомпьютеров случае, когда ее компоненты не имеют общих средств синхронизации. Если в примере на рис. 5.1 ПЭ объединены общей памятью кластера, то в ней может быть отведен участок памяти - память асинхронного канала - в котором процессы смогут хранить и забирать данные. Память асинхронного канала также делится роллингом и в некоторые моменты времени t_0 и t_1, $t_0 < t_1$, блоки данных могут размещаться там так, как показано на рис. 5.8. Такой канал может также служить средством для реализации асинхронного MPI.

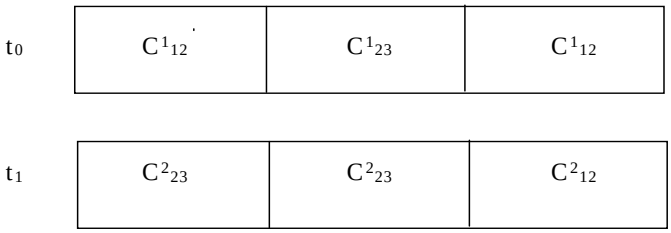


Рис. 5.8

Чтобы процессы не конкурировали между собой, как обычно, старший процесс имеет преимущество при захвате памяти асинхронного канала и называет своего единственного очередного преемника на доступ к этой памяти. Преемник должен будет в языке параллельного программирования СПП указываться специальным оператором. Таким образом можно будет запрограммировать различные порядки обменов данными между процессорами, например такие;

a) $c^1_{12}, c^1_{23}, c^1_{34},\ldots, c^1_{(n-1)n}, c^2_{12}, c^2_{23},\ldots, c^2_{(n-1)n}$

b) $c^1_{12}, c^2_{12}, c^1_{23}, c^2_{23},\ldots$

c) $c^1_{12}, c^1_{23}, c^2_{12}, c^1_{34}, c^2_{23},\ldots$

Роллинг какого-либо ресурса не всегда эффективен. Например, если процессы занимают ПЭ существенно разное время и нет между ними взаимодействий, необходимо вместо

роллинга ПЭ использовать схему назначения процесса на первый освободившийся ПЭ. В языке параллельного программирования по умолчанию может для распределения ресурсов использоваться, к примеру, схема роллинга, но при необходимости она должна заменяться другой, что указывает программист специальным оператором. Более того, для каждого вида ресурсов может выбираться свой М-алгоритм, именно так реализован выбор М-алгоритма в СПП Иня [10].

Понятно, что если каждый ПЭ связан с двумя соседними, то в такой мультикомпьютер по схеме роллинга эффективно могут быть реализованы лишь алгоритмы с m-окрестностью при m=0,1 (m=0, когда взаимодействий между операциями алгоритма нет). Если же ПЭ связаны общей памятью, то в m-окрестности может быть реализована любая схема взаимодействий, класс эффективно реализуемых на мультикомпьютерах алгоритмов существенно расширится.

5.5. Система параллельного сборочного программирования Иня

Можно указать на специфические особенности параллельного программирования крупноблочных мультикомпьютеров. Во-первых, в соответствии с их архитектурой весь ввод/вывод данных осуществляется управляющей(-ими) ЭВМ и, следовательно, программа должна

использовать максимально выделенные ей ресурсы, динамически, в ходе вычислений, настраиваться на доступные ресурсы: оперативную память, устройства и каналы ввода/выводы.

Вторым динамически захватываемым ресурсом являются ПЭ. Каждая программа должна захватить необходимое количество ПЭ. В-третьих, параллельная программа в Ине собирается из модулей - заранее подготовленных фрагментов вычислений, причем программа для ПЭ собирается динамически непосредственно перед загрузкой ПЭ. В зависимости от количества ресурсов ПЭ (памяти данных и программ) в очередной ПЭ загружается соответствующее количество данных и формируется (собирается из модулей) нужная программа обработки.

Понятно, что разные исполнения исходной параллельной программы даже с одними и теми же данными, на одном и том же мультикомпьютере породят, вообще говоря, разные системы параллельно протекающих процессов в зависимости от доступных в момент исполнения ресурсов. Такая динамическая настройка на доступные ресурсы является важнейшей характеристикой системы параллельного программирования в условиях, когда структура и конфигурация мультикомпьютера строго не фиксированы, поскольку прикладная параллельная

программа должна эффективно исполняться на любом собранном крупноблочном линейном иерархической мультикомпьютере.

5.5.1. Основные компоненты СПП

СПП Иня предназначена для программирования иерархических мультикомпьютеров и состоит из двух основных компонент. Первый - система динамического распределения ресурсов, составляющая своеобразный “ассемблерный” уровень параллельного программирования. Второй - язык и система программирования на Ине.

Система управления вычислительными ресурсами может быть реализована в двух формах: в форме децентрализованной управляющей программы и централизованного монитора.

5.5.2. Децентрализованное управление

Управляющая программа применяется в первую очередь в мультикомпьютере с простой радиальной структурой связей между управляющей ЭВМ и ПЭ (рис. 5.9). Для захвата ПЭ используется обычный для операционных систем метод управления ресурсами посредством очередей. Из своей программы, написанной на Фортране или С или другом подобном языке, пользователь обращается к управляющей программе с помощью операторов-подпрограмм. Одна из них - **INIT**, позволяет задаче захватить конкретное число ПЭ (десять,

например) либо некоторое число ПЭ, например, в интервале (2,8), то есть, допустим захват любого числа ПЭ, но не меньше двух и не больше восьми. Все ПЭ захватываются в монопольное владение.

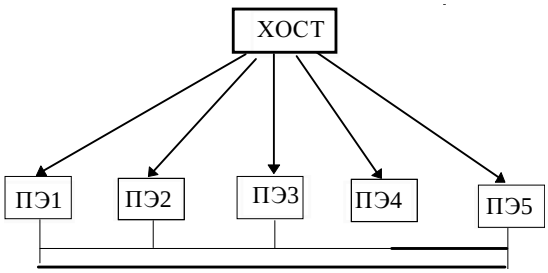


Рис. 5.9.

Подпрограмма **SET** объявляет текущим один из доступных ПЭ. Для него может быть сформирована работа (программа и данные), загружена в ПЭ и он начнет ее выполнять. После этого другой ПЭ может быть объявлен текущим, загружен работой и т.д. Если все ПЭ загружены работой, управляющая ЭВМ может заняться другой полезной работой. Подпрограмма **WAIT** позволяет управляющей ЭВМ дождаться завершения работы одного конкретного ПЭ или всех захваченных задачей ПЭ. А подпрограмма **WD** позволяет дождаться завершения работы одного, первым закончившим из всех захваченных и работающих ПЭ. Этот ПЭ загружается немедленно новой работой. Подпрограмма **RELEASE** освобождают один или все ПЭ.

Рассмотрим небольшой пример параллельной программы, вычисляющей скалярное произведение 10 пар векторов длины 1000 с вещественными компонентами. Векторы задаются двумя матрицами *V*(1000,10) и *W*(1000,10). Предполагается, что параллельная программа выполняется в многозадачной ОС и несколько параллельных программ могут исполняться одновременно, конкурируя и разделяя динамически процессорные элементы. Каждый ПЭ вычисляет скалярное произведение одной пары векторов.

Текст программы записан со сдвигом от начала строки, комментарии к программе пишутся с начала строки. Оставляя в стороне несущественные для понимания детали, параллельная программа выглядит так (используется вызов подпрограмм из Фортрана).

Вначале, как обычно, описываются и инициализируются все необходимые переменные.

REAL *V*(1000,10), *W*(1000,10), C(10)

C(i), i=1,...,10, хранит скалярное произведение *V*(*, i), *W*(*, i),

...

Программа должна быть в состоянии исполняться на любом числе процессорных элементов от двух до 10.

CALL INIT (*PNUMB, ACTION, STATUS*)

Захватываются доступные ПЭ. Значение PNUMB равно (2,10), представление пары чисел в Фортране не обсуждается. *PNUMB* определяет, что программа запрашивает ПЭ числом не меньше 2-х и не более 10.

Если мультикомпьютер имеет только 5 ПЭ (рис. 5.9), то 10 ПЭ никогда и не будет захвачено, однако на другом мультикомпьютере это может произойти и выполнение программы ускорится. При захвате ПЭ происходит следующее. Каждый ПЭ является отдельным ресурсом ОС. Программа **INIT** запрашивает ПЭ1. Если ПЭ1 свободен, он захватывается **INIT** в монопольное использование. Поскольку минимально необходимого количества ПЭ - 2-х - еще не захвачено, программа **INIT** запрашивает ПЭ2. Если ПЭ2 занят, тогда запрашивается ПЭ3 и т.д. до последнего ПЭ (в нашем случае до ПЭ5). Если захвачено 2 или более ПЭ, то программа **INIT** нормально завершает работу. Если нет, то допускаются два варианта поведения программы **INIT.**

Она может завершить работу и сообщить в главную программу, что минимально требуемое число ПЭ не удалось захватить. Тогда в главной программе может быть принято решение выполнить другую работу, а затем через некоторое время повторить попытку захвата ПЭ в надежде, что они освободились.

Другой вариант - программа **INIT** становится в очередь к некоторому произвольно выбранному занятому ПЭ и ждет его освобождения. Это решение - расплата за простоту реализации децентрализованного управления. В нем нет центра, где скапливалась бы информация, на основе которой можно было бы принять решения о том, какой ПЭ освободится другой программой раньше всех. И нельзя встать в ожидание освобождения какого-то (любого) ПЭ. В случае неудачного выбора ожидаемого ПЭ возможна ситуация, когда программа ожидает освобождения ПЭ3 тогда, когда ПЭ5 уже давно свободен. Выбор одного из этих двух вариантов поведения программы **INIT** определяется значением параметра *ACTION*.

Значение параметра *STATUS* сообщает главной программе результат исполнения **INIT**, в том числе и количество *PN* захваченных ПЭ. Все захваченные ПЭ нумеруются (логически) последовательно ПЭ1, ПЭ2, ... , ПЭ*n*, $n \leq PN$. Эти логические номера могут быть использованы в программе для назначения некоторой работы на конкретный ПЭ. Физические номера ПЭ в программе недоступны.

Следующий фрагмент программы - это цикл по числу пар векторов. В теле цикла формируется программа для ПЭ, пересылается в память ПЭ, туда же пересылаются входные данные, затем ожидается завершение вычислений в ПЭ, забирается из ПЭ в управляющую ЭВМ результат.

DO 50 *K*=1,10

J=**MOD**(*K*, *NP*)+1

Подпрограмма **MOD** вырабатывает значение kmod(NP), таким образом $1 \leq j \leq NP$.

CALL SET (*J*)

Текущим ПЭ объявлен процессор с номером *J*. Все последующие подпрограммы **PUT**, **PRODUCT**, **WR**, **GET** - не немедленно исполняемые, они дишь формируют работу для ПЭ и начнут реально исполняться только по команде **START**.

CALL PUT (*V*(*, *K*), *IV*, ...)

CALL PUT (*W*(*, *K*), *IW*, ...)

Пересылаются в ПЭJ значения *K*-ых столбцов матриц *V* и *W* в массивы *IV* и *IW* памяти ПЭ соответственно.

CALL PRODUCT (*IV*, *IW*, *IC*, ...)

Вычисляется скалярное произведение *k*-ой пары векторов.

CALL WR

Ожидание завершения счета в ПЭ

CALL GET (*IC*, *C(K)*)

Значение *K*-го скалярного произведения пересылается из ПЭ в управляющую ЭВМ

```
Вот теперь программа работы ПЭ сформирована и
ее можно запустить на счет
```

50 **CALL START**

Сформированная программа ПЭ выполняется без участия управляющей ЭВМ средствами ПЭ. Поэтому управляющая ЭВМ после запуска программы **START** начнет исполнение следующей итерации цикла, чтобы загрузить работой следующий ПЭ.

5.5.3. Централизованное управление

Централизованный монитор имеет все описанные выше подпрограммы. Организация монитора другая и функций он может выполнять больше. В момент старта монитор захватывает монопольно все доступные ПЭ, а затем выделяет их прикладным задачам по их запросам. Если запрошенное число свободных ПЭ имеется в наличии, монитор выделяет их задаче, а если нет - ставит запрос в очередь, которая обслуживается по известным дисциплинам. За счет централизованного сбора информации о состоянии ресурсов мультикомпьютера не возникает неоправданных задержек программ. Кроме того гарантируется, что стартовавшая задача получит все необходимые ресурсы и сможет исполниться, минимизируются простои как программ, так и ПЭ.

Запросы монитора устроены сложно, можно сформулировать из задачи требование на предоставление ПЭ, связанных каналами в кольцо, или на кластер, потребовать, чтобы в каждом выделенном ПЭ память данных была, к примеру, не менее 64 Мбайт и т.п.

Монитор имеет очень важную для эксплуатации многокомпонентного мультикомпьютера подсистему обеспечения надежности вычислений и живучести всего мультикомпьютера. Для поддержки работоспособности мультикомпьютера монитор периодически либо по аварийным ситуациям проводит тестирование компонент мультикомпьютера разной сложности, своевременно выводит из состава мультикомпьютера дефектные устройства и сообщает об ошибках. Каждой задаче по ее запросу монитор выдает работоспособные на момент удовлетворения запроса ресурсы мультикомпьютера.

5.5.4. Язык и система параллельного программирования Иня

Для реализации линеаризованных алгоритмов разработаны язык и система сборочного параллельного программирования Иня. Язык Иня - это расширение Фортрана, в Фортран добавлены операторы **PARDO, PAREND**, операторы для работы с ресурсами мультикомпьютера и описания

взаимодействий между процессами, а также набор вспомогательных подпрограмм подготовки данных для процессов.

Операторы **PARDO** и **PAREND** определяют участок программы, в котором формируется процесс: готовятся данные процесса (вводятся, переформируются), создается программа для ПЭ (набирается из готовых подпрограмм), организуются вычисления (захватываются ПЭ, загружаются работой, выполняются взаимодействия между процессами, выгружаются результаты и т.д.).

Программа в Ине компонуется из готовых подпрограмм так же, как и сам мультикомпьютер. Язык Иня предназначен не столько для определения вычислений, сколько для их организации. Конкретные вычисления (модули, процедуры) задаются с использованием систем программирования, существующих в математическом обеспечении вычислителей из состава мультикомпьютера.

Взаимодействия между процессами определяются в Ине оператором **PASS**, ожидания - оператором **WAIT**, точки захвата и освобождения ресурсов и требования на них указываются операторами **EID, BOD, SECT, POST**. Предусмотрено также большое число процедур для преобразования данных, подготавливаемых для передачи в ПЭ. При исполнении параллельной программы Иня обеспечивает совмещение

всевозможных работ, разделение таких ресурсов мультикомпьютера, как ПЭ и оперативная память управляющей ЭВМ, настройку на доступные ресурсы мультикомпьютера.

Структура коммутации крупноблочного иерархического линейного мультикомпьютера фиксирована, и это позволяет обеспечивать переносимость прикладных программ, написанных на Ине также тем, что оператор **PASS**, определяющий взаимодействие между процессами, реализуется по разному в зависимости от имеющихся в мультикомпьютере на момент исполнения параллельной программы средств реализации межпроцессорных взаимодействий: каналов, общая шина или общая электронная память. Поэтому прикладные программы, в которых обмен реализуется только через каналы с синхронизацией процессов в момент обмена, будут исполняться без изменений и в мультикомпьютере, в которой возможен обмен через общую память, причем обмен будет выполнен уже асинхронно.

Чтобы разобраться со средствами программирования языка ИНЯ, рассмотрим несколько различных параллельных программ решения одной и той же задачи.

Пусть необходимо поэлементно суммировать два вектора большой длины *MAX*, расположенные на внешнем устройстве. Векторы делятся на отрезки равной длины, которые

суммируются параллельной в ПЭ. Число элементов вектора, обрабатываемых в одном процессе (длина отрезка), обозначается *MIN*, *MIN* << *MAX*. Длина отрезка выбирается таким образом, чтобы поместиться в памяти одного ПЭ. Число отрезков, на которые разбиваются векторы для обработки, а следовательно, и число параллельных процессов, равно *M=MAX/MIN*.

Для ввода/вывода данных в/из управляющей ЭВМ используются входные/выходные буферы в оперативной память управляющей ЭВМ. Каждый входной/выходной буфер содержит все входные/выходные данные процесса. Исполнение программы в управляющей ЭВМ образует процесс с номером 0, остальные процессы нумеруются 1, 2, ... , *k*. Там, где это необходимо, мы будем в примерах для определенности предполагать, что мультикомпьютер имеет 3 ПЭ.

а).Пусть допускается использование только одного входного и одного выходного буферов. Тогда программа имеет вид:

DIMENSION *A(MIN), B(MIN), C(MIN)*

A и *B* - векторы-слагаемые, *C* - вектор-сумма, *BUF* содержит три отрезка векторов длины *MIN* - два входных отрезка и один выходной. Таким образом, массив *BUF* может рассматриваться как один входной и один выходной буферы.

M = *MAX/MIN* + 1

Число процессов равно числу отрезков

SET 1, *M*

Оператор **SET** требует захватить от 1 до *M* ПЭ

PARDO *M, I*

Оператор **PARDO** определяет начало цикла, число итераций цикла равно *M, I* - параметр цикла, изменяющийся от 1 до *M, I*-я итерация цикла формирует *I*-ый процесс. Захватывается ПЭ для исполнения *I*-ого процесса.

READ (10'*I*) *A*

Считываются *I*-ые отрезки векторов в память управляющей ЭВМ

PASS 0, *A, I, AAP, MIN*

Оператор **PASS** пересылает данные из массива *A* старшего процесса (с номером 0) в *I*-й подчиненный процесс в массив *AAP*, размещенный в памяти ПЭ. Количество передаваемых элементов массива равно *MIN*. Оператор **PASS** не исполняется немедленно, он включается в программу формируемого процесса и будет исполнен после ее запуска.

READ (20'*I*) *B*

PASS 0, *B, I, BAP, MIN*

CALL *VADD (AAP,1,BAP,1,CAP,1,M)*

Процедура *VADD* вычисляет поэлементную сумму векторов *A* и *B*, результат C содержится в массиве *CAP* в памяти ПЭ. *VADD* не исполняется немедленно, процедура *VADD* включается в программу формируемого процесса.

WAITR

Оператор **WAITR** ожидает завершения счета *VADD* в ПЭ, он тоже не исполняется немедленно, а включается в состав программы процесса.

PASS *I, CAP*, 0, *C, MIN*

Оператор **PASS** пересылает результат из *I*-го подчиненного процесса в старший процесс.

EXEC

Сформированная программа процесса запускается на счет в этом ПЭ. Далее может продолжиться формирование программы процесса на фоне счета ее первой части в ПЭ.

WAITP

В управляющей ЭВМ ожидается конец счета в ПЭ, чтобы продолжить дальнейшую обработку результата в управляющей ЭВМ

WRITE (30'*I*) *C*

Отрезок результирующего вектора записывается на внешнее устройство

PAREND

Происходит переход к формированию программы (*I*+1)-го процесса

END

Так как имеется только по одному неразделяемому входному и выходному буферу, то процессы выполнятся строго последовательно, хотя и будут последовательно использованы несколько ПЭ. Схематически исполнение процессов на 3-х ПЭ показано на временной диаграмме (рис. 5.10).

I-ый сформированный процесс захватывает очередной ПЭ по схеме роллинга и исполняется на нем, остальные ПЭ простаивают, так как они не могут ввести свои входные данные. Таким образом, исполнение программы происходит фактически последовательно.

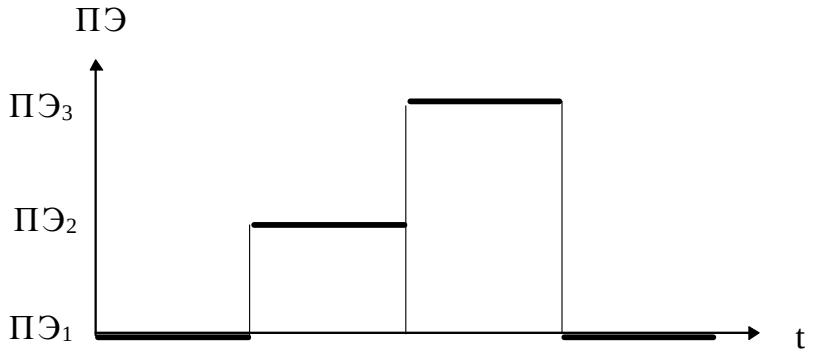


Рис. 5.10.

б) С использованием операторов **EID** и **BOD**, указывающих особые точки процессов, эта же задача может быть решена значительно более эффективно. В этом примере оператор **EID** отмечает точку в программе формирования процессов, после которой входной буфер процессом освобождается, а **BOD** указывает, что с этого момента выходной буфер захвачен процессом. Теперь можно реально организовать вычисления на нескольких ПЭ с использованием лишь одного входного и одного выходного буферов.

Начало программы совпадает с программой примера а). Памяти управляющей ЭВМ хватает для размещения только для одного входного и одного выходного буферов. Каждый процесс начинает исполнение с захвата необходимых для его

исполнения ресурсов, включая ПЭ, входной и выходной буферы.

PARDO *M,I*

READ (10'*I*) *A*

READ (20'*I*) *B*

PASS 0, *A, I, AAP, MIN*

PASS 0, *B, I, BAP, MIN*

Считываются во входной буфер входные данные формируемого процесса и пересылаются в память ПЭ.

EID

Как только *I*-й процесс исполнит оператор **EID,** (*I*+1)-й процесс сможет использовать входной буфер для ввода своих данных в память управляющей ЭВМ

CALL *VADD (AAP, 1, BAP, 1, CAP, 1, M)*

Программа *VADD I*-го процесса загружается в ПЭ для исполнения, но не исполняется.

WAITR

PASS *I, CAP, 0, C, MIN*

EXEC

WAITP

WRITE (30'*I*) *C*

BOD

Как только *I*-й процесс исполнит оператор **BOD**, (*I*+1)-й процесс сможет использовать выходной буфер.

PAREND

Выполнение фрагмента программы от **PARDO** до **PAREND** происходит следующим образом. Вначале вводятся входные данные *I*-го процесса в память управляющей ЭВМ и вводятся в память $ПЭ_i$. Затем формируется программа *I*-го процесса и вводится в память $ПЭ_i$. Сформируется следующая результирующая программа *I*-го процесса в памяти ПЭ (понятно, что вместо параметров заданы их значения):

PASS 0, *A, I, AAP, MIN*

PASS 0, *B, I, BAP, MIN*

EID

VADD (AAP, 1, BAP, 1, CAP, 1, M)

WAITR

PASS *I, CAP, 0, C, MIN*

По оператору **EXEC** программа *I*-го процесса запускается на счет, а программа управляющей ЭВМ "зависает" в ожидании на операторе **WAITPR**. Программа *I*-го процесса начинается с выполнения операторов **PASS**, пересылающих входные данные *I*-

го процесса в ПЭ$_i$. Затем исполняется оператор **EID**. Оператор сообщит управляющей ЭВМ, что I-й процесс освободил входной буфер, он может быть использован (*I*+1)-м процессом для ввода своих данных. Когда программа *I*-го процесса завершит выполнение оператора **EID**, начнется исполнение программы *VADD* в ПЭ$_i$, а параллельно с этим управляющая ЭВМ начнет исполнение (*I*+1)-й итерации цикла **PARDO-PAREND**. При этом незавершенная *I*-я итерация составляет ожидающий процесс, который продолжит исполнение после выполнения условия ожидания.

Далее, (*I*+1)-я итерация цикла **PARDO-PAREND** введет входные данные (*I*+1)-го процесса, сформирует программу (*I*+1)-го процесса в памяти ПЭ$_{i+1}$. Предположим, что выходной буфер еще занят *I*-м процессом. Будет сформирована только частичная программа, потому что выходной буфер к этому моменту не может быть захвачен:

PASS 0, *A, I+1, AAP, MIN*

PASS 0, *B, I+1, BAP, MIN*

EID

Оператор **EID** стартует выполнение программы (*I*+1)-го процесса в памяти ПЭ$_{i+1}$. Параллельно с

работой *I*-го процесса (*I*+1)-й процесс введет свои входные данные из памяти управляющей ЭВМ в память $ПЭ_{i+1}$. Если к этому моменту выполнение *I*-го процесса в $ПЭ_i$ не завершилось, то (*I*+1)-й процесс выполнит оператор **EID** и зависнет на ожидании освобождения выходного буфера *I*-м процессом. Параллельно может стартовать (*I*+2)-я итерация цикла **PARDO-PAREND**, которая начнет работу по формированию (*I*+2)-го процесса. Когда *I*-й процесс исполнит оператор **BOD**, тогда может продолжиться приостановленное исполнение (*I*+1)-я итерация цикла. Продолжится формирование (*I*+1)-го процесса, в памяти $ПЭ_{i+1}$ будет доформирован следующий фрагмент программы (*I*+1)-го процесса:

VADD (AAP, 1, BAP, 1, CAP, 1, M)

WAITR

PASS *I+1, CAP, 0, C, MIN*

и начнется исполнение этого фрагмента в $ПЭ_{i+1}$.

Каждый процесс p_i программы состоит из фрагментов in^1 - ввод данных в память управляющей ЭВМ, put^1 - передача входных данных из памяти управляющей ЭВМ в память ПЭ, run^1 - исполнение программы в ПЭ, get^i - передача результата из памяти ПЭ в память управляющей ЭВМ, out^i - пересылка данных

из памяти управляющей ЭВМ во внешнюю память. Временна́я диаграмма (рис. 5.11) исполнения системы процессов может выглядеть так:

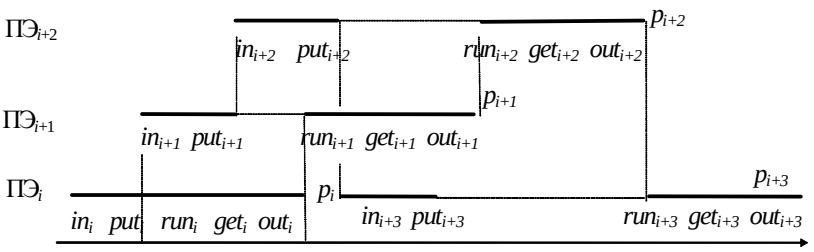


Рис. 5.11.

Как видим, наличие одного входного и одного выходного буферов в памяти управляющей ЭВМ уже делает возможным параллельное исполнение системы процессов (конечно при условии, что время работы ПЭ заметно больше времени ввода/вывода входных/выходных данных процессов).

в) С использованием операторов **SECT** - захватить некоторый ресурс - и **POST** - освободить ранее захваченный ресурс, разделение ресурсов может быть выполнено еще более эффективно.

Первая половина программы совпадает с программой примера б).

Памяти хватает только для одного входного и одного выходного буферов.

PARDO M, I

SECT

Некоторый ресурс захвачен, здесь на самом деле имеется ввиду входной буфер.

I-ый процесс продолжает выполнение, вводя свои входные данные в захваченный буфер. (*I*+1)-ый процесс должен будет здесь задержаться, т.к. нет другого экземпляра этого ресурса.

READ (10' *I*) *A*

READ (20' *I*) *B*

PASS 0, *A, I, AAP, MIN*

PASS 0, *B, I, BAP, MIN*

POST

Входные данные пересданы в ПЭ, входной буфер свободен и его можно освободить для использования другим процессом.

CALL *VADD (AAP, 1, BAP, 1, CAP, 1, M)*

CALL *APWR*

SECT

Захвачен выходной буфер

PASS *I, CAP, 0, C, MIN*

Как только *I*-й процесс пройдет *SECT, (I+L)*-й процесс может выполняться до оператора *SECT*, используя тот же процессор

EXEC

WRITE (30' *I*) *C*

POST

Выходной буфер освободился

PAREND

Структура коммутаций крупноблочного иерархического линейного мультикомпьютера фиксирована и это позволяет обеспечивать переносимость прикладных программ, написанных на Ине также тем, что оператор **PASS,** определяющий взаимодействие между процессами, реализуется по разному в зависимости от реализации межпроцессорного интерфейса: каналов, общая шина или общая электронная память. Поэтому прикладные программы, в которых обмен реализуется только через каналы с синхронизацией процессов в момент обмена, будут исполняться без изменений и в мультикомпьютере, в котором возможен обмен через общую память, причем обмен будет выполнен уже асинхронно.

VI. ОТОБРАЖЕНИЕ АЛГОРИТМОВ НА РЕСУРСЫ МУЛЬТИКОМПЬЮТЕРА

Одной из наиболее сложных проблем организации параллельных вычислений на мультикомпьютере, от качества решения которой существенно зависит качество исполнения параллельной программы, является распределение ресурсов вычислителя. Один из вариантов динамического решения этой проблемы был рассмотрен в предыдущей главе.

6.1.Статическая задача

Проблема формулируется как задача отображения алгоритма в/на ресурсы мультикомпьютера. Предполагается, что программа представляется множеством программ процессов, связанных передачами данных (взаимодействиями), и изображается ориентированным графом процессов (рис. 6.1.а). Если структура коммутационной сети мультикомпьютера образует, к примеру, двумерную решетку, то массовый алгоритм на рис. 6.1.а мог бы быть отображен в мультикомпьютер так, как это показано на рис. 6.1.б. Для обеспечения конвейерного исполнения массового алгоритма каждая операция назначается на исполнения на отдельный процессор, причем непосредственно информационно зависимые операции должны быть назначены на процессоры, связанные физическими линками (коммутационными каналами).

Параллельная программа составляется из фрагментов, каждый из которых определяет программу выполнения процесса.

Эти фрагменты загружаются в процессорные элементы мультикомпьютера в соответствии с отображением M и запускаются на исполнение. Компилятор системы программирования должен уметь формировать процессы, строить отображение M и определять загрузку программ процессов в ПЭ мультикомпьютера в соответствии с этим отображением. При переносе параллельной программы на другой мультикомпьютер (с другим числом ПЭ и другой коммутационной сетью) отображение M должно строиться заново. Таким образом, именно алгоритмы конструирования отображения M в первую очередь ответственны за обеспечение переносимости параллельной программы.

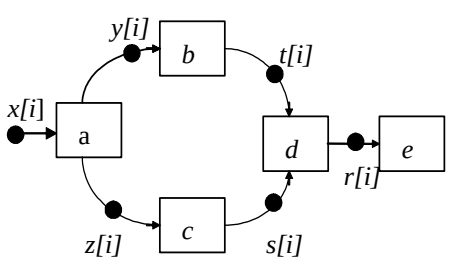


Рис. 6.1.а

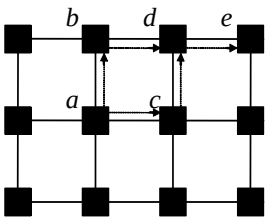


Рис.

Отображение M может сопоставлять некоторому ПЭ (назначать на ПЭ) более одного процесса.

Для достижения хорошего качества параллельной программы отображение M должно обладать рядом полезных свойств.

Свойство 1. Взаимодействующие процессы должны отображаться в один и тот же ПЭ либо в соседние ПЭ (связанные физическими линками).

Свойство 2. Все ПЭ должны быть загружены работой примерно одинаково, одновременно начинать работу и одновременно ее завершать.

- **Замечание 1.** Как обычно, качество отображения *M* оценивается некоторым функционалом. Время выполнения параллельной программы *T*, потребляемые ресурсы *R* мультикомпьютера могут служить примерами такого функционала. Свойства 1 и 2 одновременно выполнимы с приемлемым качеством далеко не всегда. Всегда, однако, существуют отображения, удовлетворяющие свойствам 1 и 2, но мало пригодные в практике. Например, отображение всех процессов в один ПЭ существует всегда. Практической пользы от этого отображения немного. Распараллеливаются обычно большие программы и ресурсов одного ПЭ, как правило, недостаточно для ее исполнения. Другая крайность, когда для исполнения конкретной программы собирается мультикомпьютер, в котором для выполнения каждой операции алгоритма есть отдельный процессор, а для реализации каждого взаимодействия - свой неразделяемый линк. Такое отображение также удовлетворит обоим условиям, но для реализации алгоритма понадобится слишком

много ресурсов. На практике чаще всего имеется конкретный мультикомпьютер и его надо оптимально использовать, хотя для решения конкретных важных задач (классов задач) могут собираться конкретные мультикомпьютеры с нужным количеством ресурсов.

Свойство 3. Каждый ПЭ должен иметь достаточное количество ресурсов, необходимых для исполнения всех назначенных на него процессов.

- **Замечание 2**. Свойство 3 ограничивает минимальное количество ресурсов, нужных для выполнения программы. Однако при конструировании параллельной программы, реализующей алгоритм, экономия ресурсов вычислителя обычно не является первоочередным критерием. Если программе минимально требуется для исполнения 3 ПЭ, но на 10 ПЭ она исполняется в 20 раз быстрее, то последнее отображение и может оказаться наилучшим в конкретном случае. Как правило, задача конструирования подходящего отображения M является сложно решаемой многокритериальной задачей, с которой (не слишком хорошо) справляются применением различных эвристик.
- **Замечание 3.** Выполнение свойства 1 нужно для того, чтобы уменьшить затраты времени на межпроцессные коммуникации. Если это свойство не выполняется для двух процессов, то пересылка данных из одного такого процесса в

другой пойдет транзитом через промежуточные ПЭ, что связано обычно с большими затратами ресурсов и значительной потерей времени.

- **Замечание 4.** Худший случай невыполнения свойства 2 тот, когда n ПЭ начинают одновременно исполнять загруженные в них процессы и недолгое время t_1 работают параллельно, затем $n-1$ ПЭ останавливаются (например, для размещенных в них процессов не выработаны еще входные данные), а один ПЭ еще долгое время t_2 работает. В завершение все n ПЭ работают параллельно в течении короткого времени t_3. Если обозначить абсолютный простой ПЭ $O=(n-1)*t_2$, а абсолютную полезную загрузку мультикомпьютера $W=nt_1+t_2+nt_3$, то понятно, что относительная загрузка мультикомпьютера $WL=W/(O+W)$ стремится к $1/n$ с увеличением t_2 при небольших t_1 и t_3.

  Освободить $n-1$ простаивающих ПЭ на время t_2 для выполнения другой работы с тем, чтобы затем их снова захватить, нельзя ввиду опасности дедлока как следствие попытки дозахвата ресурса. Поэтому простой $O=(n-1)*t_2$ практически неизбежен. Понятно, что накопить безопасно (дозахватить по частям, удовлетворяя условие банкира) $n-1$ процессорный элемент невозможно без неограниченно больших потерь ресурсов мультикомпьютера. Действительно, в момент, когда программа p_1 попробует дозахватить нужные

ей *n-1* ПЭ, ещё несколько параллельных программ $p_2,\ldots, p_m$ могут исполняться на мультикомпьютере одновременно, разделяя его ресурсы. Для такого дозахвата ПЭ программа p_1 вначале должна получить максимальный приоритет по доступу к ПЭ, чтобы никакая другая программ не могла оспорить её право на освободившиеся ПЭ, после чего переходит в состояние ожидания свободных ПЭ. Если одна из исполняющихся программ p_r, $2 \le r \le m$, завершилась и освободила свои k_r ПЭ, эти k_r ПЭ захватываются программой p_1. Если $k_r < n$, то p_1 остается в состоянии ожидания до завершения следующей программы. И так до тех пор, пока таким накопительным способом не наберёт нужные ей *n-1* ПЭ. Понятно, что это ожидание может быть сколь угодно большим. Но конечно, как это часто бывает в программировании, существует теоретически такой крайний случай, когда $t_2 >> t_1+t_3$, и в момент дозахвата ни одна из программ $p_2,\ldots, p_m$ не владеет ПЭ. На реальной смеси задач (в обычном мультипрограммном режиме эксплуатации мультикомпьютера) этот случай мало вероятен.

- **Замечание 5**. Свойства 1 и 2 выполнимы одновременно далеко не всегда. На рис. 6.2.а изображен граф операций массового алгоритма, а на рис. 6.2.б - структура коммуникационной сети мультикомпьютера. Если предположить, что время выполнения всех процессов

примерно одинаково, то для отображения M (рис. 6.2.б) процесс d должен передавать данные транзитом через ПЭ b и c, а следовательно и программа будет выполняться с большими затратами времени и ресурсов на коммуникации. При неудачно сконструированном отображении M коммуникационные затраты «съедают» весь выигрыш от распараллеливания задачи, и не исключено даже увеличение времени параллельного выполнения программы по сравнению с последовательным исполнением на однопроцессорном компьютере.

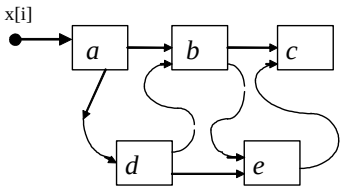


Рис. 6.2.а

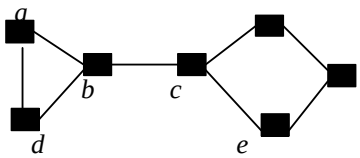


Рис. 6.2.б

Задача статического (до начала вычислений) конструирования отображения M не имеет хорошего общего технологичного решения и требует таких знаний об алгоритмах (к примеру, время исполнения процесса), которые часто не известны до начала вычислений и даже нередко не известны в ходе вычислений (например, время выполнения итерационных процессов).

- **Замечание 6**. Время работы каждого процесса может зависеть от входных данных. Например, в случае, когда в

программе содержится итеративный счет или обработка записей файла, число которых заранее неизвестно. Следовательно, величина загрузки ПЭ статически определяется весьма и весьма приблизительно. Понятно, что такая неопределенность может привести (и приводит) к неудачным решениям

- **Замечание 7**. Пусть в программе есть циклический участок, считывающий и обрабатывающий записи файла. Для обработки каждой записи файла порождается отдельный процесс. Число итераций порождающего процессы цикла зависит от размера файла и не известно до начала вычислений. Следовательно, количество процессов на этапе конструирования отображения M вообще неизвестно. В дополнение к этому, все порожденные процессы могут иметь существенно разное время выполнения, а конкретные времена исполнения процессов неизвестны и не могут быть известны. По этой причине при конструировании отображения M приходится пользоваться нечёткой информацией вида "время выполнения операции a значительно больше времени выполнения операции b". В условиях такой неопределенности вообще не приходится ожидать конструирования отображения M хорошего качества.
- **Замечание 8**. В случае, когда мультикомпьютер содержит в своем составе не универсальные (специализированные) ПЭ, не

всякий процесс может быть назначен на исполнение на любой ПЭ. Такого сорта ограничения на комбинаторные задачи построения расписаний [11, 12] обычно самым негативным образом сказываются на качестве построенного отображения *M*.

6.2. Статическое построение отображения *M*.

Существует множество различных постановок задачи построения расписаний. Здесь рассмотрен простой её вариант в приложении к планированию заданий с учетом требуемых ресурсов[1].

Пусть заданы m боксов (*bin*) B_1, B_2, …, B_m, каждый объёмом c. Пусть также даны n единиц хранения p_1, p_2, …, p_n, объем каждого p_i, $1 \le i \le n$, обозначим $r(p_i)$. Предполагается, что $\forall i$ $r(p_i) \le c$. Все p_i надо так разместить в B_i, чтобы некоторый функционал Φ достиг экстремума[2].

Можно сформулировать 3 проблемы:

P1. **Для фиксированного c найти минимальное число боксов m, в которые могут быть упакованы все p_i.** Это, например, случай планирования загрузки мультикомпьютера с целью так минимизировать число используемых для решения задачи

[1] E.Coffman, J.Leung, D.Ting. *Bin-Packing Problems and their Applications in Storage and Processor Allocation.* Computer Performance, K.M.Chandy and M.Reiser (eds), North Holland Publishing Company, 1977, p. 327-339

[2] Такие задачи ещё называются задачами построения *расписаний*

процессоров, чтобы успеть её решить к заданному моменту времени.

Р2. **Для фиксированного m найти минимальное c такое, что все p_i могут быть упакованы в m боксов.** Это случай, когда надо найти минимальную производительность процессорных элементов мультикомпьютера с m процессорами, достаточную для решения задачи в заданное время.

Р3. **Для фиксированных m и c найти максимальное $n' \leq n$ такое, что n' единиц хранения могут быть упакованы в m боксов.** Это случай, когда имеющийся мультикомпьютер должен быть максимально использован (задача *пакетной обработки*).

Здесь B_i трактуются как процессорные элементы мультикомпьютера, c – объём имеющихся в каждом B_i ресурсов, p_i – процессы, которые должны быть исполнены на мультикомпьютере и запрашивают для своего исполнения некоторое количество ресурсов $r(p_i)$. Каждое конкретное назначение p_i на B_i задаёт отображение M (расписание). Допускаются отображения, в которых суммарный запрос ресурса в каждом B_i не превышает c.

Все три проблемы NP-полны [13] и для их решения должны использоваться алгоритмы нахождения приближенного решения.

Общая схема решения таких *переборных* задач состоит в построении множества всех допустимых (удовлетворяющих ограничениям задачи) расписаний, вычислении на каждом построенном расписании s функционала $\Phi(s)$ и нахождении такого расписания s_0, на котором функционал $\Phi(s_0)$ достигает экстремального значения. Вообще говоря, может быть найдено не одно такое расписание.

Рассмотрим сначала алгоритм построения множества всех допустимых расписаний для случая $m=n$. Каждое расписание представляется n-кой (набором) вида $(p_1,p_2,\ldots,p_n)$, где p_1 в первой позиции набора означает назначение процесса p_1 на исполнение на процессор B_1 и так далее. Пусть на качество допустимых расписаний не накладываются никакие ограничения, на каждый процессор назначается только один процесс. В число допустимых расписаний попадает, например, $(p_1,p_1,\ldots,p_1)$. Тогда все множество допустимых расписаний строится тем же алгоритмом прибавления единицы, каким строятся все n-ричные числа от нуля до n^n-1. Число всех допустимых расписаний в данном случае равно n^n.

В программировании обычно $m>>n$. Поэтому надо разрешать назначение на один процессор нескольких процессов

(но не более m) и тогда множество допустимых расписаний увеличится, так как в каждой позиции набора появится ещё возможность перебора и станут допустимыми расписания типа ($\{p_1,p_2\}$, $\{p_3,p_4\}$, …,$\{p_{n-1},p_n\}$). Множества допустимых расписаний, конечно, очень велики и на практике работать с ними нельзя.

Ограничения задачи могут значительно сократить множество допустимых расписаний. Для параллельных программ естественным является ограничение, чтобы каждый процесс был назначен не более чем на один процессор[3], при этом расписания типа ($p_1,p_1,\ldots,p_1$), ($\{p_1,p_2\}$, $\{p_1,p_3\}$,…,$\{p_{n-1},p_n\}$) становятся не допустимыми. Другое ограничение связано с учетом информационных зависимостей. Если она существует, например, между процессами p_1 и p_2, и процессоры B_1 и B_2 соседние (связаны физическим линком), а процессоры B_1 и B_3 не являются соседними, то расписания типа ($\{p_1,p_2\}$, $\{p_3\}$,…,$\{p_n\}$) и ($\{p_1\}$, $\{p2,p3\}$,…,$\{p_n\}$) допустимы (информационно зависимые процессы назначены на один или соседние процессоры), а расписание ($\{p_1\}$, $\{p3\}$, $\{p2\}$,…,$\{p_n\}$) – не допустимо. Такие сокращения множества допустимых расписаний очень полезны для уменьшения затрат на получение решения, так как отбраковка расписания происходит ещё на этапе его конструирования, часть расписаний не строится вообще, и значение функционала не вычисляется.

[3] Ограничение естественное, если не надо обеспечивать высокую надежность вычислений. В противном случае оно недопустимо

К сожалению, добавление новых ограничений при попытке сократить множество допустимых расписаний нередко быстро приводит к отсутствию решения (не существует расписания, удовлетворяющего всем ограничениям). В таком случае пытаются ослабить какие-то ограничения. Например, разрешить транзитные коммуникации в 2-окрестности процессора, затем в 3-окрестности и так до тех пора, пока не найдется приемлемое решение. Понятно, что множество допустимых расписаний при этом расширяется. Поэтому на практике задача решается эвристическими алгоритмами, которые строят приемлемое расписание в соответствии с некоторой стратегией.

Такие стратегии используют некоторое дополнительное знание о задаче. Рассмотрим следующий пример. Пусть заданы m идентичных процессоров B_1, B_2, …, B_m. Каждый процессор B_i, $1 \le i \le m$, имеет оперативную память размером $|B_i|$. Заданы информационно независимые процессы p_1, p_2, …, p_n, здесь $p_j:(m_j,t_j)$, каждый процесс p_j, $1 \le j \le n$, требует для своего исполнения память объемом m_j, время исполнения t_j. Необходимо построить (субоптимальное) расписание с приемлемым временем исполнения. Несколько процессов могут быть назначены в процессор B_i, если их суммарный запрос памяти не превышает $|B_i|$.

К примеру, расписание ($\{p_1, p_2, p_3\}$, $\{p_5, p_9\}$, $\{p_{n-5}\}$, …,$\{p_n\}$) показывает, что назначенные на процессор B_1 процессы $\{p_1, p_2, p_3\}$ будут и исполняться в порядке перечисления, а именно: сначала исполнится процесс p_1, затем, по его завершении, стартует процесс p_2, и затем только выполнится процесс p_3.

Стратегия конструирования расписания должна учесть свойства задачи. Для сформулированной задачи ясно, что нехорошо оставлять назначение процессов, требующих для своего исполнения большого объема памяти, на конец – они могут не поместиться в памяти никакого процессора из-за того, что часть памяти процессоров уже занята ранее назначенными процессами. Так же нехорошо откладывать в конец назначение долго исполняющихся процессов – может получиться так, что все прочие процессы закончили исполнения, а последний процесс ещё долго будет работать, задерживая завершение всего множества процессов. В обоих случаях могут построиться неудачные расписания. Потому разумно для построения расписания использовать такую стратегию:

- Строятся два списка процессов:
  - L1 - список процессов, упорядоченных по убыванию объёма запрашиваемой памяти
  - L2 - список процессов, упорядоченных по убыванию времени исполнения,

- Процессы из первой трети списков L1 и L2 (они содержат процессы с самыми большими запросами ресурсов) назначаются на процессоры, более или менее равномерно заполняя память и потребляя время процессоров.
- Затем аналогично назначаются процессы из второй, а затем и последней трети списков L1 и L2.

Построенное расписание на практике может быть и очень хорошим и не очень, но вряд ли оно будет совсем уж плохим.

Определить понятие приемлемое (хорошее) расписание не просто, однако часто можно построить оценку стратегии и доказать, насколько построенное решение будет отличаться от оптимального, например, будет хуже оптимального не более чем на 30%. Решения в пределах этих 30% и могут считаться приемлемыми.

Это практически полезная и часто используемая стратегия, которая позволяет строить приемлемые расписания. Её часто применяют и в жизни. Если надо оптимально упаковать чемодан (максимально использовать его пространство), то практичные люди сначала укладывают в чемодан самые крупные и самые тяжёлые вещи, а затем все остальное, и получается неплохо. Однако, если размеры и вес укладываемых вещей не соответствуют размерам и прочности чемодана, то никакая стратегия назначения не приведет к хорошему результату. А вот если загружать чемодан песком (каждая песчинка намного

меньше чемодана и все они одного размера и веса), то при любой стратегии укладки песчинок чемодан будет загружен оптимально. Так что построение хорошего расписания зависит не только (а скорее не столько) от методов построения расписания, сколько от свойств задачи (процессов, процессоров).

Можно сделать еще несколько существенных замечаний, но и из приведенных примеров видны трудности использования статических методов конструирования отображения *M* в системах параллельного программирования. Потому статические методы конструирования отображения *M* в широкой практике становятся все менее интересными, хотя для них есть важные специальные приложения. В последнее время статические методы в параллельном программировании больших численных моделей вытесняются динамическими, которые на практике оказались значительно более технологичными. А потому далее обсуждаются методы динамического конструирования отображения *M*. Один динамический способ конструирования отображения *M* уже обсуждался в главе 5. Другой способ будет демонстрироваться далее на примере параллельной реализации метода частиц в ячейках (*Particle-In-Cell, PIC*) в сборочной технологии параллельного программирования мультикомпьютеров.

6.2.Идеи параллельной реализация PIC

Сборочная технология параллельного программирования разрабатывалась для поддержки параллельной реализации численных моделей большого размера в физике [5]. Прежде чем обсуждать общие принципы сборочной технологии параллельного программирования мультикомпьютеров, рассмотрим конкретный пример её применения для решения конкретной задачи – разработки параллельной программы, реализующей метод частиц в ячейках (кратко – метод частиц либо *PIC – Particle-In-Cell*) для крупномасштабного моделирования обмена энергией в облаке плазмы. Это позволит накопить конкретный материал для обсуждения.

Рассмотрим вначале вычислительную суть метода частиц (структуру вычислений, а не математическую модель) и проблемы его параллельной реализации, а затем обсудим способ распараллеливания алгоритмов реализации *PIC*. Детальное математическое описание метода частиц в его разных приложениях можно найти в [14].

6.2.1.Краткое описание метода

При моделировании физического явления методом частиц исходное физическое пространство представляется в компьютере *пространством моделирования* (ПМ). Часто ПМ имеет форму прямоугольного параллелепипеда (рис. 6.3).

Для дискретизации значений электрического *E* и магнитного *B*

полей все пространство моделирования разбивается прямоугольной сеткой на ячейки. В вершинах ячеек определяются значения электрического *E* и магнитного *B* полей (*сеточные значения*). На самом деле, для обеспечения вычислительной устойчивости счета, дискретизация полей производится более сложным образом (см. рис. 6.4), но для наших целей эта деталь не очень важна и её обсуждение, как и многих других, опускается. Чем меньше шаг сетки, то есть чем больше ячеек, тем точнее представление полей.

Плазма представляется множеством модельных частиц, характеризующихся координатами, массой, скоростью и величиной заряда. Частицы распределены в пространстве моделирования в соответствии с некоторым законом, который изменяется во времени. Частицы влияют друг на друга, изменяя значения электромагнитных полей. Допускается наличие частиц существенно разного рода (ионы, электроны) и они обрабатываются разными алгоритмами.

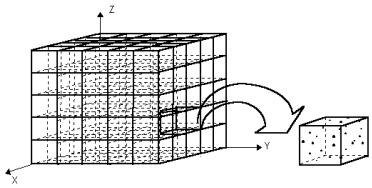


Рис. 6.3

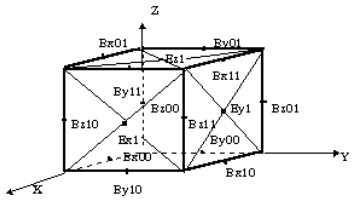


Рис. 6.4

С каждой ячейкой связаны частицы, находящиеся в этой

ячейке в данный момент времени. Электромагнитные поля, заданные в вершинах ячейки, действуют на частицы внутри нее, и за малое время Δt их скорости и координаты изменяются (частицы передвигаются в новую точку ПМ).

В свою очередь «перелетевшие» за время Δt в новую точку ПМ заряженные частицы влияют на значения электрического и магнитного полей в вершинах ячейки. Для учета этого влияния заряды частиц внутри ячейки интерполируются в вершины ячейки, таким способом вычисляются значения средней плотности тока и заряда в узлах сетки. Эти значения используются для пересчета значений электромагнитных полей.

Процесс моделирования состоит из серии временных шагов. На каждом шаге по времени:

1) на основе текущих значений электромагнитных полей вычисляется сила, действующая на частицу;
2) вычисляются новые скорости и координаты частиц;
3) из новых координат и скоростей частиц вычисляются значения средней плотности тока и заряда в узлах сетки;
4) из полученных значений средней плотности тока и заряда в узлах сетки, пересчитываются значения электромагнитных полей.

Количество шагов моделирования определяется условиями

физического эксперимента. Так как координаты частиц изменяются в процессе моделирования, то время от времени частицы «перелетают» из ячейки в ячейку. Для устойчивости решения размер временно́го шага выбирается таким образом, чтобы в течение одного шага моделирования частица улетала не далее соседней ячейки.

6.2.2. Особенности параллельной реализации метода частиц

Обычно задачи, решаемые с использованием метода частиц, характеризуются большим объемом данных и вычислений. Например, в одном из простейших экспериментов для моделирования процесса обмена энергией в облаке плазмы использовалось 12 миллионов частиц (а надо несколько миллиардов для изучения особенностей поведения плазмы), размер вычислительной сетки - 64×64×96 (а надо бы 1000×1000×1000). Только для хранения этих данных требуется 400 Мб памяти. На каждом временно̀м шаге для вычисления новых скоростей и координат одной частицы требуется порядка ста арифметических операций. До того момента, когда можно наблюдать интересные физические результаты, требуется провести более 1000 временных шагов, то есть произвести не менее $12*10^{11}$ арифметических операций. И это очень маленькая лабораторная задача. Реальное моделирование потребует произвести многократно большие вычисления, представляя плазму несколькими миллиардами частиц, а электромагнитные

поля – большими сетками.

В алгоритме реализации PIC заложен большой внутренний параллелизм. Так как частицы между собой непосредственно не взаимодействуют, то действия, связанные с перевычислением координат и скоростей частиц можно проводить независимо друг от друга.

При решении задач на мультикомпьютерах возникает вопрос - каким образом распределить данные между ПЭ. Так как в каждый момент времени частица находится в некоторой ячейке и для обработки частицы используются сеточные значения, то для уменьшения коммуникационных издержек в процессе моделирования необходимо так распределять данные, чтобы сеточные значения, заданные в вершинах ячейки, хранились в том же ПЭ, что и атрибуты частиц.

Чтобы удовлетворить этим условиям можно:

- дублировать массивы всех сеточных значений (во всех узлах сетки) в памяти каждого ПЭ. В этом случае частицы равномерно распределяются между ПЭ, вне зависимости от их расположения в ПМ. В этом случае, однако, на каждом временно́м шаге потребуется обмениваться массивами сеточных значений между всеми ПЭ. Да и память ПЭ используется нерационально из-за дублирования больших массивов данных, что быстро приведет к ее исчерпанию и катастрофическому

уменьшению размера решаемой задачи.

- применить пространственную декомпозицию - разрезать ПМ на подобласти по числу ПЭ. В этом случае массивы сеточных значений также разрезаются на части (с дублированием граничных строк и столбцов). Тогда в каждый ПЭ помещаются части массивов сеточных значений и те частицы, которые находятся в соответствующей подобласти ПМ. Пространство моделирования разрезается на подобласти таким образом, чтобы в каждом ПЭ находилось примерно одинаковое количество частиц. Таким образом, в начальный момент времени достигается хороший баланс загрузки ПЭ.

Однако в процессе моделирования частицы изменяют свое местоположение в ПМ, перелетая из ячейки в ячейку. Для того, чтобы частицы по-прежнему находились в том же ПЭ, что и соответствующие ячейки, иногда приходится передавать атрибуты частицы из одного ПЭ в другой. Таким образом, в процессе моделирования может накопиться большой дисбаланс загрузки ПЭ - в некоторые ПЭ прилетит много бóльше частиц, чем в другие (возможно что и все). Поэтому необходимо время от времени производить *динамическую балансировку* загрузки ПЭ (*перебалансировку*), выравнивая число частиц в процессорах мультикомпьютера.

Нужно сказать, что даже начальная сбалансированная загрузка не всегда может быть сконструирована, как это справедливо для случая моделирования взрыва облака плазмы. В этой задаче все частицы в исходный момент сконцентрированы внутри одной ячейки и, следовательно, должны размещаться в одном ПЭ, что невозможно для задачи большого размера - все частицы попросту не поместятся в памяти одного ПЭ, хотя памяти всех ПЭ достаточно для решения задачи.

6.2.3. Сборочный подход к конструированию программы

Второй подход и следует использовать для параллельной реализации метода частиц. Реализация основана на применении сборочной технологии (СТ) к конструированию программы. Основная идея СТ состоит в том, что вначале определяются неделимые (*атомарные*) фрагменты вычислений, а вся программа собирается затем из этих атомарных фрагментов. Фрагментарная структура сборочной программы сохраняется и в исполняемом коде, что позволяет при необходимости передавать фрагменты на исполнение из одного ПЭ в другой (*перелет* процесса из одного ПЭ в другой). Для метода частиц таким естественным атомарным фрагментом является ячейка (вместе с частицами и сеточными значениями, необходимыми для вычисления силы, действующей на частицу, а также со всеми вычислениями (кодом, процедурой) над этими данными). Поэтому частицы и сеточные переменные, заданные в вершинах

ячейки, всегда оказываются в одном ПЭ.

Для бóльшей конкретности дальнейших рассуждений фрагментированную программу можно представлять себе как множество атомарных фрагментов. Каждый атомарный фрагмент реализован в виде процедуры. На каждый процессорный элемент назначается на исполнение такое количество фрагментов, чтобы загрузка всех ПЭ была одинаковой. Это может быть сделано хорошо, если фрагменты невелики[1]. Напомним, что в первых трех стадиях моделирования (см. 6.1.1) вычисления ведутся в цикле по всем частицам. Время, затрачиваемое на выполнение этого цикла, составляет в зависимости от задачи от 70 до 90% всего счетного времени. Поэтому критерием загрузки процессоров в этой задаче может служить число обрабатываемых в процессоре частиц. Таким образом, если в процессоре p_0 обрабатывается N_0 частиц, а в процессоре $p_1 - N_1$ частиц, то процессоры считаются равно загруженными при выполнении условия N_1-$\varepsilon < N_0 < N_1$+ε . Пороговое значение ε определяет частоту проведения операции балансировки загрузки и выбирается специальным алгоритмом.

Программа каждого ПЭ - это главный цикл по всем фрагментам, назначенным на исполнение в этот ПЭ. В случае перегрузки хотя бы одного ПЭ, часть атомарных фрагментов передаётся на исполнение (*улетает*) в соседний менее

нагруженный ПЭ. Улетевшие процессы исключаются из главного цикла перегруженного ПЭ, а в недогруженном ПЭ прилетевшие процессы включаются в главный цикл. Для случаев типа моделирования взрыва используется техника виртуальных фрагментов, когда один реальный фрагмент представляется несколькими виртуальными фрагментами, содержащими описание одной и той же ячейки, а частицы равномерно делятся между ними.

Понятно, что такая программа может быть крайне неэффективной при большом числе маленьких фрагментов (содержащих небольшой объем вычислений)[2]. Большая часть ресурсов и времени работы мультикомпьютера тратились бы в ней на реализацию управления, поддержание фрагментированности программы и перелетов процессов. Для примера, платой за сохранение процедурной структуры последовательной программе во время её исполнения может быть 10% уменьшение её производительности по сравнению с монолитной программой. Обеспечение перелета многочисленных небольших фрагментов многократно уменьшило бы производительность всей программы.

Вообще желательно описывать сборочную программу в

---

[1] Аналог упаковки чемодана песком в предыдущей главе

[2] Да и реализация фрагментов процедурой неприемлема по той же причине.

терминах столь малых фрагментов, какие только допускаются задачей (ячейки в методе частиц), а выполнять программу с сохранением в процессе счета сравнительно небольшого числа сравеительно крупных фрагментов с тем, чтобы уменьшить накладные расходы на реализацию управления в программе. Поэтому вводится дополнительно понятие *минимального фрагмента*, объединяющего несколько атомарных фрагментов. Минимальные фрагменты конструируется статически перед началом вычислений как высокоэффективные процедуры, свои для каждого типа мультикомпьютера. В ходе конкретного исполнения программы минимальные фрагменты неделимы.

6.3.Распараллеливание метода частиц

В распараллеливании метода частиц в ячейках в качестве атомарного фрагмента всегда используется ячейка, т.е., все переменные и программы обработки данных внутри ячейки. В качестве минимального фрагмента выбираются разные подмножества ячеек в зависимости от структуры коммуникационной сети мультикомпьютера.

6.3.1.Распараллеливание метода частиц для линейки ПЭ (линеаризация *PIC*)

В качестве минимального фрагмента используется слой ПМ «толщиной» в одну ячейку. Всё ПМ разбивается на блоки (рис. 6.5), каждый блок состоит из нескольких прилегающих слоёв. Блоки назначаются на ПЭ, число слоёв в каждом блоке выбирается таким образом, чтобы загрузка всех ПЭ была бы

примерно одинакова (дисбаланс не превышал некоторого допустимого порога). В случае превышения порога дисбаланса хотя бы в одном ПЭ (слишком велика или слишком мала загрузка), он инициирует процедуру балансировки и подходящий граничный слой (граничные слои) блока должен перелететь на менее загруженный соседний ПЭ.

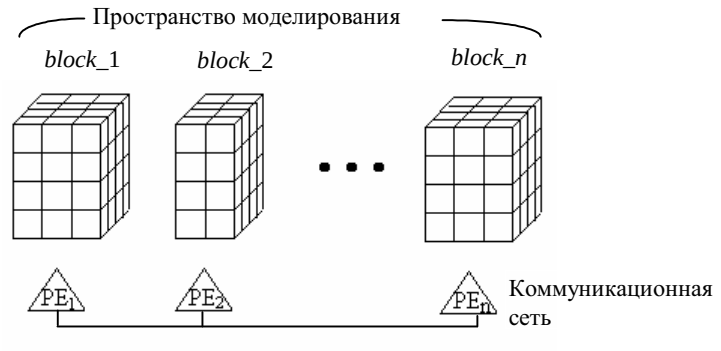


Рис. 6.5

6.3.2.Отображение линеаризованного *PIC* на 2D решетку ПЭ

Колонка ячеек выбирается в качестве минимального фрагмента для мультикомпьютеров со структурой коммуникационной сети типа 2*D* решетки и/или гиперкуба.

Для обеспечения исполнения линеаризованного PIC на решетке ПЭ строится его отображение на решетку. Пусть дана $l \times m$ 2D решетка ПЭ. Блок *block_i* (рис. 6.6) назначается для обработки на i-ю строку 2*D* решетки ПЭ, $1 \le i \le l$. Понятно, что *block_i* формируется таким образом, чтобы обеспечить равную

загрузку каждой строки ПЭ мультикомпьютера. При этом, например, строка ПЭ, в которой один или более ПЭ неработоспособны, получит пропорционально меньшую нагрузку. Далее, каждый *block_i* делится на *m* подблоков *block_ij*, *j*=1,...,m, (по числу ПЭ в строке процессоров), которые распределяются на исполнение среди ПЭ строки. В свою очередь блоки *block_ij* формируются таким образом, чтобы обеспечить равную загрузку каждого ПЭ строки (рис. 6.6).

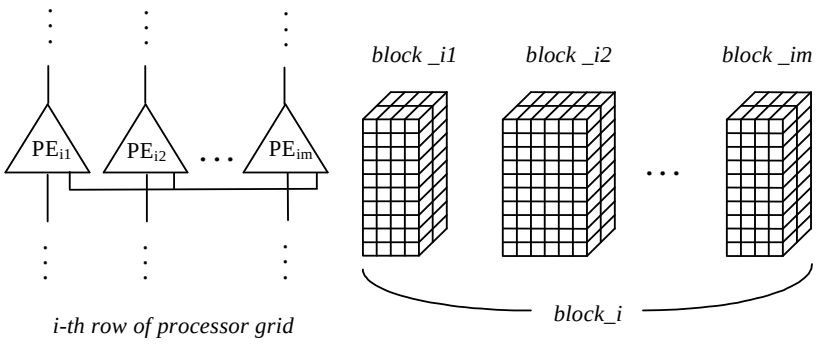


Рис. 6.6

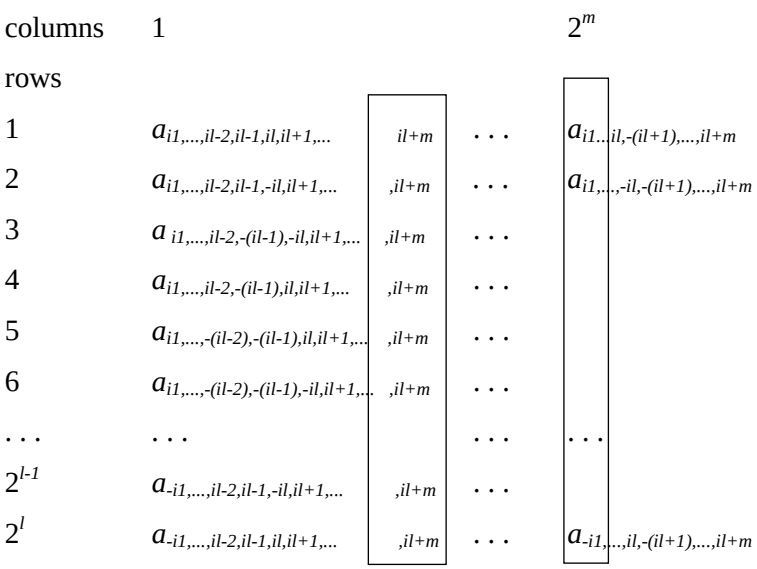
6.3.3. Отображение 2*D* решетки ПЭ на гиперкуб

Коммуникационная структура гиперкуба также хорошо подходит для исполнения линеаризованного *PIC*. Линеаризованный *PIC* отображается в гиперкуб следующим образом.

Обозначим узлы *n*-гиперкуба $a_{i1,i2,...,in}$, $ik=\{1,-1\}$, $1 \le k \le n$. Каждый узел $a_{i1,i2,...,in}$ соединён с *n* другими узлами, чьи номера отличаются от номера узла $a_{i1,i2,...,in}$ только в одной позиции

индексов, т.е. их номера имеют вид *i1,..., -ik,....,in* для некоторого *k*, $1 \le k \le n$.

Пусть мы используем $2^l \times 2^m$ решетку, $l+m=n$. Тогда соответствие между узлами гиперкуба и решетки таково:



Номера узлов гиперкуба в каждой строке имеют одни и те же значения первых *l* индексов. Номера узлов в каждой колонке имеют одно и то же значение последних *m* индексов. При этом номер каждого узла решетки отличается от номера соседнего узла только в одном индексе, а значит эти узлы соединены физическим каналом. Часть каналов гиперкуба вообще не используются.

6.4.Централизованные алгоритмы балансировки загрузки

При реализации *PIC* на линейке ПЭ в качестве минимального фрагмента выбирается слой ячеек с тем, чтобы сохранить регулярность структур - хранить сеточные значения по-прежнему в массивах и производить вычисления в цикле над массивами. В начальный момент времени слои распределяются между ПЭ таким образом, чтобы в каждом ПЭ было примерно одинаковое количество частиц. Смежные слои назначаются в один и тот же либо в соседние ПЭ. Смежные слои, размещенные в одном ПЭ, образуют блок.

6.4.1. Начальная балансировка загрузки ПЭ

Эвристический алгоритм конструирования начальной балансировки загрузки. В каждом ПЭ строятся два массива:

- *A* содержит данные о количестве частиц в каждом слое ПМ (*A[i]* - количество частиц в *i*-м слое);
- в *S* заносятся данные о распределении слоев между ПЭ (*S(i)* - номер первого слоя блока, размещенного в $ПЭ_i$).

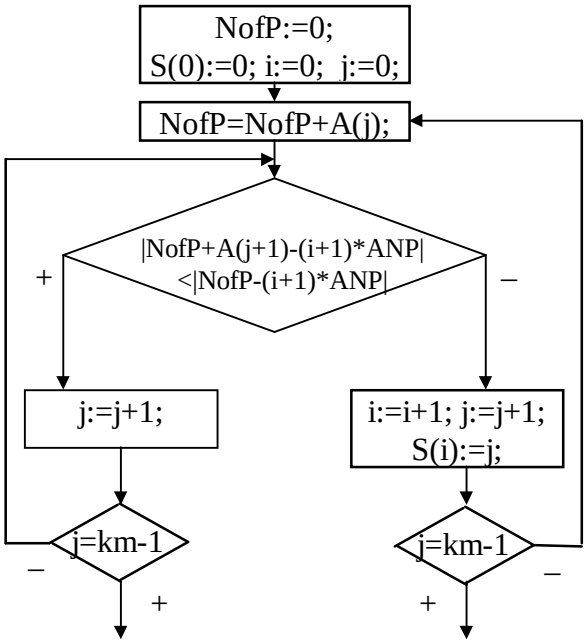


Рис. 6.7

Таким образом, каждый ПЭ «знает», какие слои назначаются в другие ПЭ, это, собственно, *карта распределения нагрузки* между ПЭ.

Алгоритм имеет сложность *O(km)*, где *km* - количество слоев ПМ. Обозначим общее количество частиц как *NP*, число ПЭ - как *NPE*, среднее количество частиц в каждом ПЭ *ANP=NP/NPE*. Нулевой слой назначается в $ПЭ_0$ (нумерация ПЭ и слоев начинается с нуля). Далее в цикле по элементам массива *A*, слой *j+1* назначается в $ПЭ_i$, если при этом количество частиц в первых

i+1 ПЭ менее отличается от *(i+1)*ANP*, чем без частиц этого слоя (рис. 6.7). Этот эвристический алгоритм строит очень хорошую начальную загрузку процессоров мультикомпьютера

6.4.2. Динамическая балансировка загрузки

Идею динамической балансировки загрузки можно сформулировать следующим образом. Пусть программа представлена как множество процессов. Тогда алгоритмы динамической балансировки загрузки мультикомпьютера должны обеспечивать поведение множества процессов в мультикомпьютере аналогичное поведению жидкости в системе сообщающихся сосудов, а именно: множество процессов должна «переливаться» из перегруженных ПЭ в недогруженные ПЭ так же, как жидкость переливается из переполненного сосуда в менее заполненный. Алгоритмы балансировки загрузки должны так же сглаживать пики нагрузки ПЭ, как сила притяжения сглаживает волны на поверхности воды в пруду (небольшие волны остаются).

Если взять мультикомпьютер с топологией коммуникационных межпроцессорных связей типа двумерной решётки и в какой-то момент выполнения программы нарисовать график функции нагрузки его процессорных элементов, он и должен напоминать водную рябь на поверхности пруда. Высота ряби зависит от величины порога дисбаланса нагрузки. Такие алгоритмы динамической балансировки загрузки хорошо еще и тем, что программа может вообще ничего не знать о структуре коммуникационной сети (в ее коде эта структура никак не отражается). В каждом процессоре лишь хранится информация о соседних процессорах, все взаимодействия локальны. Это

свойство исключительно важно при программировании мультикомпьютеров с большим числом ПЭ (тысячи), при обеспечении устойчивости программы к сбоям (поломка) ПЭ и/или линков коммуникационной сети.

На основе этих аналогий естественным кажется вывести универсальные алгоритмы динамической балансировки из разностной схемы для двумерного уравнения диффузии.

Построенный алгоритм покажется красивым и обоснованным, но в чистом виде он не работоспособен. Все-таки множество процессов программы – это не жидкость и ведут процессы себя по-другому. Рассмотрим несколько примеров, когда алгоритм не работает должным образом.

- **Гора песка**. Пусть программа исполняется на мультикомпьютере с решеткой ПЭ размером, к примеру 16×32, и пусть нагрузка ПЭ возрастает вдоль широкой части решётки от $ПЭ_{1,1}$ до $ПЭ_{1,32}$, дисбаланс нагрузки между соседними ПЭ не превышает, но близок к порогу дисбаланса нагрузки ε. Тогда дисбаланс нагрузки между $ПЭ_{1,1}$ и $ПЭ_{1,32}$ близок к 32ε. Если учесть, что для рассмотренной задачи моделирования поведения облака плазмы методом частиц порог дисбаланса должен быть около 5%, то дисбаланс нагрузки в 160% между крайними процессорами $ПЭ_{1,1}$ и $ПЭ_{1,32}$ представляется неприемлемым. Система процессов ведет себя как гора

песка, а не как жидкость, из-за большого «трения» во множества процессов. И надо сильно «трясти» мультикомпьютер (заложить это в алгоритм динамической балансировки), чтобы «песок» процессов равномерно распространился по всем ПЭ мультикомпьютера.

- **Кольчуга**. На самом деле, из-за информационных зависимостей между процессами ситуация с балансировкой загрузки ещё хуже, чем в примере «гора песка». Ведь балансировка загрузки должна обеспечивать короткие межпроцессные коммуникации (желательно не далее соседних ПЭ). Поэтому система информационно зависимых процессов ведёт себя при исполнении как кольчуга: стоит только потянуть одно колечко (перелететь процессу в соседний ПЭ), и оно потянет за собою всех своих соседей, а те потянут за собою своих соседей и так далее. Обеспечить близость коммуникаций и равномерную загрузку ПЭ мультикомпьютера в этих условиях очень непросто. В общем случае алгоритм динамической балансировки загрузки должен тогда знать статическую структуру межпроцессных коммуникаций исполняемой программы и уметь это знание использовать. Впрочем, для задач численного моделирования это обычно простая проблема.
- **Предсказание направления диффузии**. Динамическая балансировка загрузки – не дешевая

операция и желательно минимизировать их количество. Для этого в реализации метода частиц на каждом временно́м шаге подсчитывается вектор движения частиц и каждая балансировки делается «с запасом», предвидя последующие перелеты частиц. Такое предсказание развития распределения нагрузки (учет особенностей решения конкретной задачи) позволяет в несколько раз снизить число балансировок. Понятно, что алгоритмы балансировки, учитывающие специфику задачи, будут наиболее эффективными.

В целом, приведенная аналогия множества процессов с жидкостью далека от правды. Это скорее образное описание задачи, чем способ её решения.

Рассмотрим теперь, как работает механизм балансировки загрузки в реализации *PIC*. Вначале каким-либо образом определяется порог разрешенного дисбаланса *BP* - максимальное количество частиц свыше среднего количества частиц, которое может обрабатываться одним ПЭ. Например, его можно задать константой. После каждого шага по времени каждый ПЭ подсчитывает количество частиц в нем. Если это количество отличается от среднего количества частиц *ANP* больше, чем на заданный порог, ПЭ сигнализирует о необходимости проведения балансировки загрузки.

Выделенный ПЭ собирает сведения о количестве частиц в

каждом слое (алгоритм балансировки централизованный), записывает данные в уже описанный выше массив *A* и рассылает этот массив всем ПЭ. Далее, алгоритмом начальной балансировки загрузки, каждый ПЭ строит массив *S*, в который заносятся данные о новом распределении слоев между ПЭ. После чего производится обмен слоями между соседними ПЭ - сначала (если необходимо) одновременно обмениваются слоями каждый $ПЭ_{2i}$ и $ПЭ_{2i+1}$, затем каждый $ПЭ_{2i+1}$ и $ПЭ_{2i+2}$.

Определение порога разрешенного дисбаланса BP. Были проведены эксперименты для *BP* равного 0% (когда расписание балансировки строилось на каждом шаге) и 5% от общего числа частиц. При *BP*=5% было проведено 2 серии экспериментов. В первой серии новая карта распределения нагрузки строилась каждый раз, когда уровень дисбаланса превышал заданный порог. Во второй серии экспериментов новая карта строилась только тогда, когда увеличивалось максимальное количество частиц по сравнению с предыдущим шагом.

В двух других сериях экспериментов *BP* вычислялся динамически из времени, затраченного на выполнение перебалансировки и времени, затрачиваемого на обработку одной частицы. Таким образом, можно подсчитать, обработке какого количества частиц эквивалентно выполнение одной балансировки. Это значение не является точным, так как время, затрачиваемое на проведение одной балансировки, может

значительно отличаться от времени, затрачиваемого на проведение другой балансировки - в зависимости от того, сколько слоев перелетает из одного процессора в другой. В таблице 2 приводится общее время работы программы моделирования разлета облака плазмы (сетка размера 20x20x30, размер облака, находящегося в центре ПМ - 106000 частиц; всего 430000 частиц) при использовании централизованного алгоритма динамической балансировки загрузки для различных значений *BP*. Как видно из результатов тестирования, алгоритм с динамически вычисляемым BP имеет лучшие характеристики. Это и понятно, этот алгоритм лучше «чувствует» дисбаланс загрузки.

Таблица 2.

	2 ПЭ	3 ПЭ	4 ПЭ	5 ПЭ	6 ПЭ	7 ПЭ	8 ПЭ
• BP=0%	847	597	439	385	306	289	252
• BP=5%	851	595	441	387	306	288	247
• BP=5% (при увел. дисб.)	847	594	438	383	306	315	272
• динам. BP	849	595	437	384	306	287	247
• динам. BP (при увел. дисб.)	845	595	436	383	307	315	247
• динам. балан-сировки нет	846	678	483	445	361		297

6.4.3. Виртуальные слои ПМ

В процессе моделирования возможна ситуация, когда большинство частиц собирается в одном слое или в одном

столбце ячеек или даже в одной ячейке. Для организации вычислений в таком случае используется понятие виртуального слоя. Слой ПМ (вычисления и сеточные значения, заданные в вершинах ячеек слоя) представляется несколькими копиями (виртуальными слоями), а множество частиц исходного слоя распределяется между этими *виртуальными* слоями и, следовательно, между ПЭ, обеспечивая тем самым их равную нагрузку. Основной недостаток такого решения заключается в том, что появляется потребность в «длинных» взаимодействиях, так как исходно смежные слои могут теперь располагаться не в соседних ПЭ.

Благодаря введению понятия виртуальных слоев в начальный момент времени достигается сбалансированная загрузка. Граничные слои дублируются в соседних ПЭ, частицы из этих слоев распределяются между ПЭ таким образом, чтобы в каждом ПЭ находилось равное количество частиц. Алгоритм начальной балансировки загрузки видоизменяется следующим образом. Слой 0, как и ранее, назначается в $ПЭ_0$. В цикле по элементам массива *A*, слой *j* назначается не только в $ПЭ_i$, но и в $ПЭ_{i+1}$, если при добавлении частиц слоя *j* в $ПЭ_i$ количество частиц в первых *i* ПЭ становится большим чем *i*ANP*. При накоплении дисбаланса в каждом ПЭ по массиву *A* строится новый массив *S*. Если новый массив *S* отличается от предыдущего, ПЭ обмениваются слоями. После чего распределяются частицы из граничных дублируемых

слоев.

6.4.4.Централизованный алгоритм балансировки загрузки при реализации *PIC* на решетке ПЭ

Если при реализации *PIC* на мультикомпьютерах с большим количеством ПЭ использовать только линейные связи между ПЭ, то во время выполнения централизованных алгоритмов динамической балансировки загрузки требуются длинные коммуникации. При использовании топологии связи типа «решетка» радиус системы значительно уменьшается.

Ячейки (вместе с частицами) назначаются для обработки в ПЭ следующим образом. Пусть имеется $l \times m$ ПЭ, образующих решетку. *Block_i* (состоящий из нескольких прилегающих слоев) назначается для обработки на ПЭ *i*-й строки решетки, $0 \le i < l$. Эти блоки формируются таким образом, чтобы обеспечить равномерную загрузку всех строк решетки ПЭ. После этого каждый *block_i* независимо от других блоков делится на *m* частей - блок_ij, $j=0,...,m-1$, которые распределяются между *m* ПЭ *i*-й строки. Эти блоки имеют форму параллелепипеда (рис. 6.8). Таким образом сохраняется регулярность структур и сеточные значения хранятся в массивах.

Динамическая балансировка загрузки происходит в два этапа. Сначала перераспределяются слои между строками ПЭ решетки. Используется описанный выше алгоритм динамической балансировки для линейки ПЭ. Затем новые блоки

распределяются между ПЭ строк.

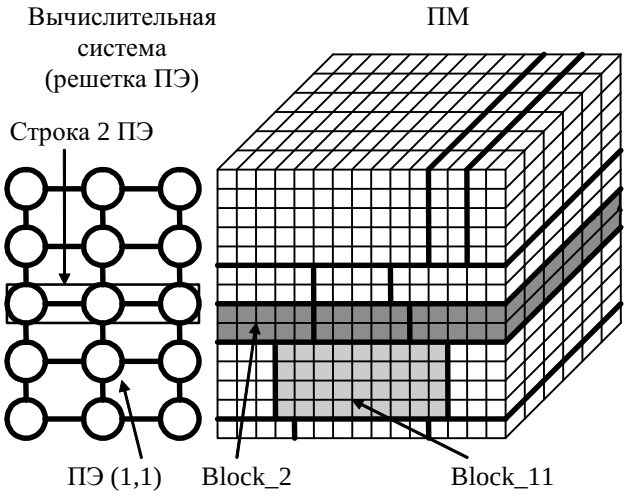


Рис. 6.8

6.5.Децентрализованные алгоритмы динамической балансировки загрузки

В таких алгоритмах решение о направлении передачи данных из более загруженных ПЭ в менее загруженные принимается локально. Цель - сбалансировать загрузку нескольких ПЭ в небольшой окрестности - локальной области. Вся вычислительная система покрывается такими локальными областями.

6.5.1. Основной диффузионный алгоритм

В основных диффузионных алгоритмах при выборе локальных областей должно выполняться условие, что нельзя разделить систему на части, не разделив при этом хотя бы одну локальную

область (бесшовное покрытие). Балансировка загрузки в области происходит лишь на основе информации о текущей загрузке ПЭ области. Диффузионные алгоритмы отличаются размерами (и формами) локальных областей. Здесь размер области ограничивается двумя ПЭ. Поведение системы во время принятия решения о направлении передачи данных синхронизовано. Принятие решения производится в несколько шагов.

На четном шаге каждая пара ($ПЭ_{2i}$, $ПЭ_{2i+1}$) ($0 \le i < N/2$, где N - количество ПЭ) обменивается информацией о количестве частиц и принимает решение о передаче части частиц из более загруженного ПЭ в менее загруженный с целью выравнивания загрузки. После чего каждый ПЭ имеет новую информацию о количестве частиц в нем.

На нечетном шаге каждая пара ($ПЭ_{2i+1}$, $ПЭ_{2i+2}$) ($0 \le i < (N-1)/2$, где N - количество ПЭ) обменивается информацией о количестве частиц и принимает решение о передаче части частиц из более загруженного ПЭ в менее загруженный с целью выравнивания загрузки.

Такая диффузия может продолжаться в течение нескольких шагов, пока, наконец, данные не будут распределены равномерно и ни в одной паре процессоров не начнется перебалансировка загрузки. В то время как в рассмотренном выше централизованном алгоритме балансировка производится за два

шага.

Тестирование алгоритма показало, что хотя при увеличении количества шагов качество балансировки улучшается, общее время работы программы изменяется незначительно.

Конечно, не все так просто с диффузионными алгоритмами. Для них, к примеру, характерна такая неприятная черта. Чистые диффузионные алгоритмы не чувствуют медленного нарастания нагрузки (гора песка). Чтобы исправить этот недостаток приходится принимать меры для пошагового распространения глобальной информации по всем ПЭ мультикомпьютера.

6.5.2. Модифицированный диффузионный алгоритм

Вычислительная система разбивается на множество смежных подобластей ПЭ. На первом шаге работы алгоритма в каждой подобласти собирается информация о количестве частиц в ПЭ этой подобласти. Это частично (на подобласти) централизованный шаг алгоритма. Далее крайние ПЭ подобластей обмениваются собранной информацией и решают, сколько частиц должно быть передано из одной подобласти в другую. Таким образом, получаем основной диффузионный алгоритм с размером локальной области равным 2, однако в качестве локальной области рассматривается не два ПЭ, а две подобласти ПЭ.

6.5.3. Децентрализованный алгоритм динамической балансировки

Предлагаемый децентрализованный алгоритм предназначен для выполнения на линейке ПЭ. Он основан на знании предметной области. Во время выполнения цикла по частицам в каждом ПЭ вычисляется значение вектора скорости, перпендикулярно которому ПМ разбивается на подобласти. Таким образом, вычисляется основное направление движения частиц. В соответствии с этим знанием смежные ПЭ решают, в какую сторону передавать лишние частицы и откуда принимать недостающие (частицы передаются в направлении, обратном основному направлению разлета).

6.6. Заключительные замечания

Частицы не являются единственным источником динамической нерегулярности данных и перегрузки ПЭ мультикомпьютера. В *PIC* методе, как и в любой численной модели, в ходе моделирования возникнет дисбаланс нагрузки ПЭ, если появляется необходимость в более точном представлении функций (*адаптивные* сетки). Например, в *PIC* потребуется делить (измельчать) ячейки ПМ для более точного представления электрических и магнитных полей, если они начинают изменяться слишком быстро в некоторой подобласти ПМ.

Другой источник нерегулярности в моделях и в PIC в частности - *переменный временно́й шаг*. Необходимость в уменьшении

временно́го шага возникает, если частица «улетает» за один шаг моделирования дальше соседней ячейки (это приведет к некорректному моделированию). Тогда временно́й шаг для этой частицы придется уменьшать, например, вдвое, что эквивалентно по нагрузке на ПЭ прилету в ячейку дополнительной частицы. Понятно, что большое число «быстрых» частиц приведет к дисбалансу загрузки ПЭ. Для медленных частиц временно́й шаг может увеличиваться. Во всех таких случаях приходится прибегать к динамической балансировке загрузки ПЭ мультикомпьютера.

Алгоритмы динамической балансировки загрузки ПЭ мультикомпьютера позволили эффективно решить одну из основных проблем реализации *PIC* и других подобных математических моделей с динамической нерегулярностью данных – проблему динамической настройки параллельной программы на изначально непредсказуемое поведение физического объекта или явления.

В нашем случае программа автоматически настраивается на складывающееся в процессе моделирования распределение частиц в пространстве моделирования, которое неизвестно, вообще говоря, до начала процесса моделирования. Моделирование ведь и затевается для изучения поведения облака плазмы. Такая настройка - необходимое и важнейшее качество параллельной программы, реализующей математическую модель

физического явления.

Интересно в заключение отметить, что поведение параллельной программы, реализующей *PIC* метод, следует в некотором смысле поведению моделируемого объекта: если частицы модели в ходе моделирования полетели в пространстве моделирования, например, вправо, то управление в параллельной программе организует перелет процессов влево в попытке сохранить сбалансированность загрузки всех ПЭ мультикомпьютера. Во множестве процессов образуются волновые процессы, зеркально отображающие волновые процессы в облаке плазмы. В конце концов становится не вполне ясно, что есть модель? То ли программа моделирует поведение облака плазмы, то ли облако плазмы является «физической» моделью поведения множества процессов параллельной программы. Следует, видимо, ожидать проявления этого свойства параллельных моделирующих программ и в моделировании других явлений. Оно кажется вполне естественным.

6.7. Общие принципы сборочной технологии параллельного программирования

Теперь можно приступить (есть необходимый материал) к обсуждению основных идей сборочной технологии параллельного программирования.

6.7.1. Собирать или делить?

Основное ключевое слово сборочной технологии

программирования – «*сборка*». Нередко в программировании используется слово «*распараллеливание*» имея при этом ввиду необходимость разделения (*partitioning*) целостно спроектированной программы (сформулированной задачи) на более или менее независимые фрагменты для последующего параллельного исполнения (декомпозиция областей в численном моделировании, например). Хорошо сделать это автоматически можно далеко не всегда, и потому в сборочной технологии используется прямо противоположный подход, при котором программа собирается из заранее заготовленных фрагментов вычислений в том же стиле, как стена складывается из кирпичей. Фрагментированная структура программы и «швы» сборки сохраняются в ходе её исполнения (в исполняемом коде).

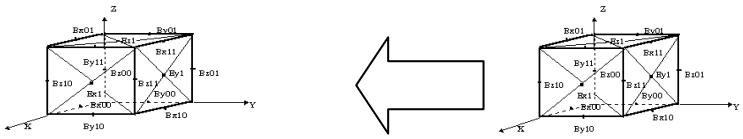
Подход к распараллеливанию (статическому или динамическому) программы теперь совсем простой: разрешается делить целую программу и данные на фрагменты только вдоль швов сборки. И сделать это можно всегда при надлежащей реализации СТ. Для примера, швы сборки в методе частиц составляют переменные на границе двух слоёв (реализация для линейки ПЭ).

6.7.2. Двухуровневая система программирования

Система сборочного программирования с очевидностью имеет двухуровневую структуру. Первый включает средства создания атомарного фрагмента программы. На этом уровне

могут использоваться все известные методы, языки и системы разработки последовательных программ.

Второй уровень составляют средства сборки программ. Специальная подсистема – *компоновщик* – формирует необходимые минимальные фрагменты из атомарных (конструирует общий неделимый код, используя коды атомарных фрагментов и учитывая алгоритм сборки), а затем и программу целиком. Алгоритм сборки описывает, какие фрагменты в каком порядке собираются и какие переменные отождествляются.



Атомарный фрагмент 1 Атомарный фрагмент 2

Рис. 6.9

Суть задачи демонстрирует пример, в котором, как обычно, опускаются многие технологически важные детали. Пусть необходимо собрать минимальный фрагмент – слой - для реализации *PIC* метода. Атомарным фрагментом является ячейка (рис.6.4.), ее код содержит простые переменные *Bz10, Bz11*, … и другие, а значит и нет циклов по этим переменным. Для большей определенности примера предположим (безотносительно к задаче), что код ячейки содержит оператор *Bz10:=Bz11+By11*;

Сборка проходит поэтапно. В системе сборочного программирования каждому атомарному фрагменту

соответствует образная 3-х мерная иконка. Иконками можно манипулировать с помощью мышки. Вначале в редакторе иконок берутся два одинаковых атомарных фрагмента 1 и 2, и один из них придвигается к другому (по стрелке на рис. 6.9) до тех пор, пока не произойдет пространственное совмещение всех переменных на боковых гранях. Если при сдвиге совмещения переменных на иконках не произошло, значит такие атомарные фрагменты не могут быть собраны в один целый объект. Переменные, которые непременно должны совмещаться при сборке определяются при описании атомарного фрагмента. Чтобы различать одноименные переменные разных атомарных фрагментов, используется верхний индекс (1 или 2) для заглавной буквы в имени переменной. Такое совмещение трактуется системой программирования как необходимость построить общий код для обработки сразу двух объединенных фрагментов, а совмещенные переменные отождествляются. Делается это следующим образом:

- описание всех переменных обоих фрагментов становятся общими
- совмещенные переменные отождествляются $B^1z11 \equiv B^2z10$, $B^1x10 \equiv B^2x00$, … и переименовываются. В частности, переменные B^1z10, B^1z11, B^2z11 в коде объединенного фрагмента описываются как массив $Bz11[n]$, $n=1,3$.

- Формируется общий код объединенного фрагмента. В частности оператор *Bz10:=Bz11+By11* преобразуется в одномерную циклическую конструкцию

```
for i=1 to 2 do
    Bz11[i]:=Bz11[i+1]+By11[i];
```

Действуя таким образом можно насобирать колонку ячеек, затем из колонок собрать слой ячеек, который и использовать далее как минимальный неделимый фрагмент вычислений.

Исполнение сборочной программы поддерживается специальной *исполнитльной системой (диспетчером)*, который расширяет функции ОС и обеспечивает перелет процессов и динамическую балансировку загрузки мультикомпьютера. В системе сборочного программирования такие динамические свойства прикладной программы как балансировка, настройка на доступные ресурсы мультикомпьютера, требуемая степень устойчивости вычислений к сбоям оборудования обеспечиваются автоматически.

6.7.3. Разделение семантики и схемы вычислений

Двухуровневая сборка программ из атомарных фрагментов приводит к тому, что семантика вычислений инкапсулируется внутри них. На уровне сборки программы конструируются лишь межпроцессные взаимодействия, управление, динамическое распределение ресурсов и другие схемные операции. И это позволяет использовать на уровне сборке методы автоматического

конструирования программ и библиотеки стандартных схем вычислений. Использование библиотек способно существенно улучшить качество автоматически сгенерированных программ.

6.7.5.Динамическая балансировка загрузки.

Фрагментированная структура сборочной программы обеспечивает реализацию диффузионных методов её динамического отображения на ресурсы мультикомпьютера и динамическую настройку на все доступные ресурсы.

Другое важнейшее свойство сборочной программы с диффузионными алгоритмами балансировки нагрузки – высокая устойчивость к сбоям оборудования. Программа просто не знает структуры коммуникационной сети (она не отражена ни в коде программы, ни в данных). Процессы программы имеют информацию только о соседних процессах, а в каждом ПЭ – информация о ближайших соседях. Все ресурсы планируются и перепланируются динамически и потому программа способна работать всегда при наличии работоспособных ресурсов и при любых изменениях их структуры и количества, сохраняющих компактность коммуникационной сети.

6.7.7.Требования к представлению массовых алгоритмов для их параллельной реализации

Теперь легко сформулировать требования к представлению массовых алгоритмов (в частности, численных алгоритмов). Таких требований немного:

- Прежде всего алгоритм должен быть достаточно мелко фрагментирован.
- Операции алгоритма должны иметь как одинаковый объем вычислений, так и одинаковый запрос ресурсов.
- Незначительная информационная зависимость между фрагментами, при которой в ходе вычислений всегда существует много готовых к выполнению фрагментов.
- Локальность межпроцессных взаимодействий,
- Минимальность объемов передачи данных

Например, явные разностные схемы позволяют так фрагментировать вычисления. В них вычисления производятся над всеми элементами матрицы и обработка каждого элемента матрица может определять атомарный фрагмент вычислений. Все элементы матрицы обрабатываются параллельно, между ними нет информационной зависимости. А вот неявные схемы менее пригодны для параллельной реализации. В них существуют такие информационные зависимости между фрагментами, что количество готовых к выполнению фрагментов часто оказывается небольшим и они не могут загрузить все ПЭ мультикомпьютера

- локальность информационных зависимостей между фрагментами.

Информационные зависимости должны связывать только соседние фрагменты, лучше всего, если все связи находятся в 1-

окрестности графа информационных зависимостей. Если существуют «длинные» связи в n-окрестности, то это может привести к времяёмким транзитным межпроцессорным коммуникациям.

7. СИНТЕЗ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ НА ВЫЧИСЛИТЕЛЬНЫХ МОДЕЛЯХ

После всех предшествующих рассмотрений настало время обсудить проблему конструирования параллельных программ в рамках подходящего формализма. В главе рассматривается метод синтеза (автоматического конструирования) параллельных программ на *вычислительных моделях* (ВМ). Рассмотрение задачи автоматического конструирования параллельных программ и обсуждение необходимых алгоритмов конструирования позволяет в строгих рамках формальной модели обсудить суть проблемы разработки параллельных программ. Большей частью глава написана по материалам книги [5].

7.1.Простые вычислительные модели

В первой главе вычислимая функция определена как функция, представимая детерминантом - конечно порожденным множеством функциональных термов. В настоящей главе рассматривается метод порождения детерминанта, основанный на известной формализации предметной области - вычислительных моделях, а также алгоритмы конструирования и оптимизирующих преобразований детерминанта.

7.1.1. Исходные соображения

В зависимости от характера формализации предметной области можно выделить несколько подходов к проблеме синтеза программ. Наиболее цельный из них в теоретическом отношении - *логический синтез* программ. При этом подходе формальным описанием предметной области является некоторое достаточно полное математическое исчисление, представляющее закономерности предметной области. Задача синтеза ставится как задача нахождения доказательства теоремы существования, а программа извлекается из этого доказательства. К сожалению, этот подход весьма сложно реализуем на практике, так как, с одной стороны, для не тривиальных предметных областей создание полной непротиворечивой теории не осуществимо. С другой стороны, еще недостаточно развиты методы машинного доказательства теорем, особенно если ищется программа, обладающая нужными свойствами, например, оптимальная по какому-либо критерию.

В конце 70-х годов в Вычислительном центре СО АН СССР началась разработка метода синтеза параллельных программ на базе вычислительных моделей. Исследования стартовали в то время, когда было объявлено о начале работ по реализации японского проекта 5-го поколения компьютеров, в основе которого лежала именно логическая модель вычислений. Причины, по которым не были достигнуты объявленные цели (в математическом обеспечении) широко разрекламированного и

весьма дорогостоящего проекта, хорошо известны. Две из них названы выше. Уже этих двух причин более чем достаточно, чтобы не питать надежд на хороший “чистый” результат и в обозримом будущем.

Эти трудности заставляют пользоваться при описании предметной области менее богатыми, но более удобными для машинного восприятия средствами. Одно из них заключается в наведении на предметной области некоторой структуры, отражающей ассоциативные связи между её понятиями. Явное выражение этих связей позволяет свести случайный поиск к ассоциативному, что решающим образом сказывается на оценках трудоемкости процесса планирования. Кроме того, структурное описание имеет наглядное графическое представление. Описанный подход к синтезу программ получил название *структурного синтеза* программ.

Проблема синтеза программ на вычислительных моделях рассматривается в статической постановке: по представлению вычислительной модели, выделенной группе переменных, служащих для ввода начальных данных, и другой группе переменных, служащих для вывода результатов, требуется найти программу, вычисляющую по заданным входным данным соответствующие результаты (если они для этих входных данных определены).

При одной и той же постановке задачи можно синтезировать различные программы. Принципиально важно, что предлагаемые алгоритмы синтезируют программу наилучшего качества. Основными критериями качества являются надежность, оптимальность (в ряде смыслов), степень параллелизма и т.п. В целом в процессе синтеза решается несколько задач, укладывающихся в следующую общую схему.

Пусть даны:

класс S спецификаций входных заданий;

класс P результирующих выходных программ;

отношение эквивалентности ~ на P;

отношение качества $>$ на P, удовлетворяющее аксиомам частичного порядка.

Требуется найти алгоритм $A{:}S{\to}P$ такой, что результат синтеза - программа $p{=}A(s)$, $p{\in}P$, - удовлетворяет спецификации $s{\in}S$ и является наилучшей в смысле $>$ в классе программ $\{p|p{\sim}A(s)\}$.

Таким образом, проблема синтеза заключается в разработке алгоритма, конструирующего по каждому элементу из S некоторый элемент из P, наилучший среди всех программ, решающих специфицированную задачу.

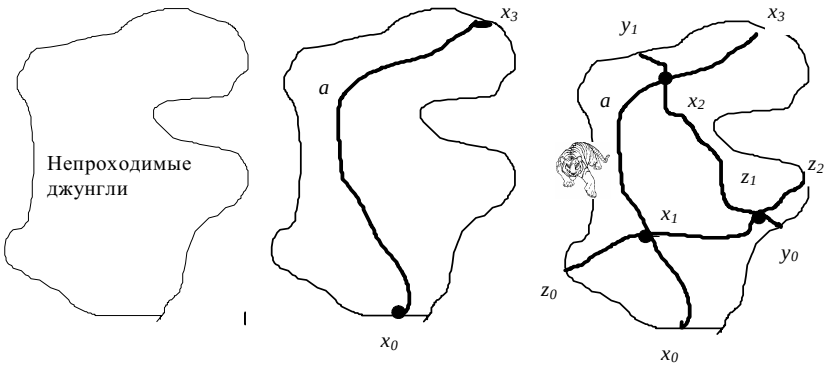
Метод синтеза параллельных программ на вычислительных моделях отличает высокая прагматичность. Прагматичность означает, что в формализме вычислительных моделей могут конструироваться приемлемые по качеству

прикладные программы, а не только демонстративные, “игрушечные” примеры типа задачи нахождения наибольшего общего делителя двух целых чисел.

Прагматичность метода основана на том, что вычислительная модель является, собственно, собранием хороших решений различных задач предметной области и решение новой задачи собирается (компонуется) из готовых хороших решений. В таких условиях есть надежда сконструировать на их основе хорошее решения новой задачи. Конечно, иногда для этого приходится включать в вычислительные модели новые необходимые решения.

Образно, идея этого подхода может быть продемонстрирована на следующем несерьезном примере. Представим себе непроходимые джунгли (рис.7.1.а). Для “автоматического” пересечения джунглей из пункта x_0 в пункт x_3 необходимо тщательное, во всех мыслимых деталях, описание джунглей (нужна достаточно полная теория в логическом программировании). На практике рассчитывать на наличие такого знания изначально невозможно, никто его и не пытается создать. Поэтому, если понадобится ходить из пункта x_0 в пункт x_3, то вначале первопроходцы, не заботясь о детальном изучении джунглей, проложат тропинку a из x_0 в x_3 (рис.7.1.б) на основе имеющихся знаний. При совершении этого подвига некоторые из них, возможно, будут съедены дикими зверями или утонут в

трясинах, но после них и все прочие люди смогут пользоваться тропинкой, имея лишь минимально необходимые знания о джунглях. Затем таким же образом могут появиться тропинки из y_0 в y_1 и из z_0 в z_2 (рис.7.1.в). Но одновременно появился и путь $x_0x_1z_1z_2$. А из x_0 в x_3 ведут теперь два пути: $x_0x_1x_2x_3$ и $x_0x_1z_1x_2x_3$. И второй, более длинный, путь вполне может оказаться предпочтительнее первого, более короткого, например в случае, когда на участке x_1x_2 путника поджидает проголодавшийся тигр.



а) б) в)

Рис. 7.1.

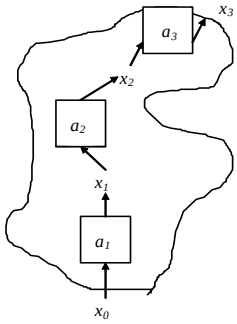


Рис. 7.2

Тропинки служат здесь образами алгоритмов. Тропинке a соответствует последовательность шагов (операций) алгоритма (рис.7.2). Операция a_1 вычисляет значение переменной x_1 из значения переменной x_0 и т. д. Операция может реализоваться в программе процедурой или фрагментом кода. Тогда переменная x_3 вычислится из x_0 последовательностью операций $x_0 \to a_1 \to x_1 \to a_2 \to x_2 \to a_3 \to x_3$. Пусть аналогично, значение переменной z_2 из значения z_0 (тропинка $z_0x_1z_1z_2$) вычисляется последовательностью операций $z_0 \to b_1 \to x_1 \to b_2 \to z_1 \to b_3 \to z_2$, а значение переменной y_1 из значения переменной y_0 (тропинка $y_0z_1x_2y_1$) — последовательностью $y_0 \to c_1 \to z_1 \to c_2 \to x_2 \to c_3 \to y_1$. Тогда переменная $x3$ может быть вычислена двумя разными алгоритмами, соответствующим двум возможным путям $x_0x_1x_2x_3$ и $x_0x_1z_1x_2x_3$, т.е. алгоритмами $x_0 \to a_1 \to x_1 \to b_2 \to z_1 \to c_2 \to x_2 \to$

$a_3 \rightarrow x_3$ и $x_0 \rightarrow a_1 \rightarrow x_1 \rightarrow a_2 \rightarrow x_2 \rightarrow a_3 \rightarrow x_3$. На этом выборе основано конструирование оптимального алгоритма.

Базой знаний в вычислительных моделях является множество алгоритмов, причем хороших алгоритмов (как тропинки в джунглях не прокладываются плохо, так и в вычислительных моделях накапливаются только хорошие алгоритмы). И комбинации хороших алгоритмов (путь $x_0x_1z_1x_2x_3$ в джунглях) тоже могут быть хороши. Они хотя и не обязательно оптимальны, но и не самые худшие. Задача вывода приемлемого алгоритма становиться простой и сводится к ограниченному управляемому перебору на графе.

В дополнению к этому, так же как массив джунглей разбиваются тропинками на фрагменты, так и описание предметной области разбивается в вычислительных моделях на множество меньших предметных областей, для которых построение более или менее полных теорий (для каждой подобласти своей) более вероятно. Таким образом, в вычислительных моделях формализованное описание предметной области строится как система теорий, связанных соотношениями модели.

В заключение необходимо также указать, что метод синтеза программ на вычислительных моделях применяется тогда, когда достаточно полная модель предметной области еще не создана, а считать уже надо - обычная на практике ситуация. С

появлением достаточно полной модели предметной области, для которой известны способы извлечения алгоритма решения задачи приемлемого качества, можно (там, где это удастся сделать) переходить к логическим методам конструирования программ как наиболее универсальным методам. Продолжая несерьезный пример, можно сказать, что сквозь джунгли, даже и по тропинкам, ходят люди опытные (хотя и разной опытности на разных этапах освоения джунглей). А вот если джунгли заасфальтировать, то тогда и автомат можно пускать: он и до пункта назначения доберется, ничего по дороге не поломает и сам себя не покалечит.

Если же говорить серьезно, то комбинированный подход имеет, видимо, наибольшие шансы на достижения успеха в автоматическом конструировании параллельных программ. Вообще необходимо заметить, что чистая теория более похожа на компас. Как и компас, она только показывает верное направление. Но ходить на практике надо по-другому, только придерживаясь в целом направления, указанного компасом, что и демонстрируют различные успешные технологии программирования. И, например, пути полярных исследователей к обоим полюсам не проходили по кратчайшим путям. Во всех случаях это были невообразимые (и необъяснимые с точки зрения теории) путаницы зигзагов, сохраняющие, однако, общее направление на полюс. Понятно, что на полярных исследователей

действовали различные прагматические факторы, не вошедшие в теорию. Такие факторы в практических предметных областях невозможно включить в теорию так, чтобы добиться её должной полноты и сохранить непротиворечивость.

7.1.2.Основные определения

Пусть заданы:

1. конечное множество $\mathbf{X}=\{x, y, ..., z\}$ переменных для представления вычисляемых и измеряемых величин;
2. конечное множество $\mathbf{F}=\{a, b, ..., c\}$ функциональных символов (операций) арности $m \times n$, $m \geq 0$, $n \geq 0$, m и n, вообще говоря, различные для разных операционных символов;
3. с каждым символом операции a арности $m \times n$ связан набор $in(a)=(x_1, ..., x_m)$ входных и набор $out(a)=(y_1, \ldots y_n)$ выходных переменных, при этом $i \neq j \rightarrow y_l \neq y_j$.

Пара $\mathbf{C}=(\mathbf{X},\mathbf{F})$ называется *простой вычислительной моделью* (ПВМ). Операция $a \in \mathbf{F}$ описывает возможность вычисления переменных $out(a)$ из переменных $in(a)$ с помощью некоторой процедуры. Модели удобно изображать графически. Графическое представление операции a показано на рис.7.3

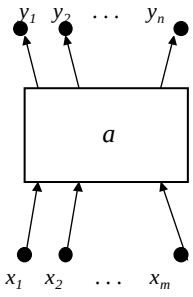


Рис. 7.3

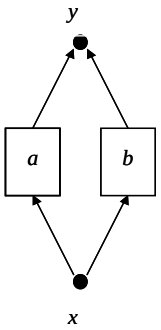


Рис. 7.4

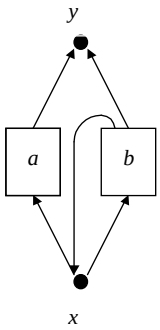


Рис. 7.5

Информационным маршрутом в модели назовем последовательность $X1, X2, ..., Xk$ такую, что

$\forall i\ ((X_i \in \mathbf{F} \cup \mathbf{X}) \wedge ((X_i \in \mathbf{F} \rightarrow X_{i+1} \in \text{out}(X_i)) \vee (X_i \in \mathbf{X} \rightarrow X_{i+1} \in \mathbf{F} \wedge X_i \in in(X_{i+1}))))$.

Маршрут вида $XX_i, \ldots, X_kX$ называется циклом. Модель называется ациклической, если не содержит ни одного цикла. Так, модель на рис.7.4 является ациклической. Простейшим примером циклической модели является модель, изображенная на рис. 7.5.

Пусть $V \subseteq \mathbf{X}$, $F \subseteq \mathbf{F}$. Определим множество термов $T(V,F)$, порожденных V и F. Для $t \in T(V,F)$ одновременно будут определяться следующие атрибуты терма:

$in(t)$ - множество входных переменных;

$out(t)$ -множество выходных переменных;

$var(t)$ - множество всех используемых переменных;

$top(t)$ - вершина терма;

oper(*t*) - множество операций;

sbt(*t*) - множество подтермов;

d(*t*) - глубина терма *t*.

Основным объектом теории является функциональный терм и многие далее рассматриваемые алгоритмы обрабатывают термы. По этой причине нам понадобится детальная терминология, связанная с описанием терма и его составных частей. Для удобства примем следующее соглашение. Если $M=(m1,...,mp)$, $N=(n1,...,nq)$ - упорядоченные множества, то

$M \cup N=\{m1, ..., mp\} \cup \{n1., ..., nq\}$,

$M \cap N=\{m1, ..., mp\} \cap \{n1., ..., nq\}$

и т.д. Таким образом, булевы операции над упорядоченными множествами будут рассматриваться как над неупорядоченными.

Формальное определение терма и его атрибутов задается следующими правилами.

1. Если $x \in V$, то x есть терм t, $x \in T(V,F)$; $in(t)=var(t)=\{x\}$; $out(t)=\{x\}$, $top(t)=x$, $oper(t)=\varnothing$, $sbt(t)$ состоит из единственного подтерма - самого терма t, $d(t)=0$.
2. Пусть t^1, ..., t^s входят в $T(V,F)$ и для t^i, $i=1,...,s$, все атрибуты определены. Пусть $a \in F$, $in(a)=(x_1,...,x_s)$. Тогда терм t, равный $a(t^1,...,t^s)$, включается во множество $T(V,F)$ тогда и только тогда, когда выполняется $\forall i(x_i \in out(t^i))$. Атрибуты терма t в этом случае определяются так:

$$in(t)=\bigcup_{i=1}^{s} in(t^i);$$

$$out(t)=out(a);$$

$$var(t)=(\bigcup_{i=1}^{s} var(t^i))\cup out(a);$$

$$top(t)=a;$$

$$oper(t)=\bigcup_{i=1}^{s} oper(t^i)\cup\{a\};$$

$$sbt(t)=\bigcup_{i=1}^{s} sbt(t^i)\cup\{t\};$$

$$d(t)=max\{d(t^i): i=1,...,s\}+1.$$

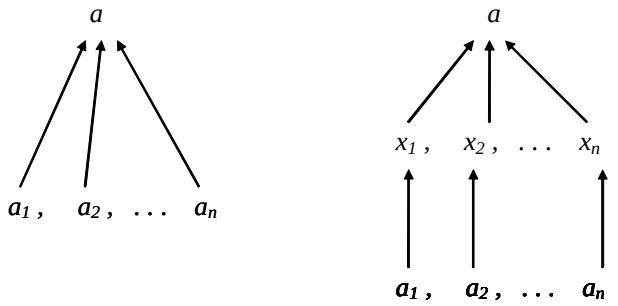
Зачастую, вместо “терм t, равный $a(t^1,...,t^s)$”, будем просто кратко писать $t=a(t^1,...,t^s)$. Это сокращение всегда легко будет отличить от предиката равенства.

Используем также следующие обозначения: символом $t'\subseteq t$ будем обозначать тот факт, что терм t' представляет собой подтерм терма t, а обозначение $t'\in t$ будем употреблять в случае, когда выполняется $t'\subseteq t\wedge t'\neq t$, то есть терм t' является собственным подтермом терма t. Кроме того, будем использовать понятие вхождения подтерма t' в терм t, связывая с ним конкретную позицию, занимаемую t' в записи терма t.

В дополнение к другим функциям определим функцию $out_t(t')$, где t - терм, t' - вхождение подтерма t' в t, не совпадающее с самим термом t. Если $t=a(t^1,...,t^s)$, $in(a)=(x_1,...,x_s)$, то $out_t(t^i)=x_i$,

i=1,...,s. Функция $out_i(t')$ показывает, какую переменную подтерм t' вычисляет в терме t. Мы будем также часто говорить, что подтерм t_i вычисляет в терме t переменную x_i.

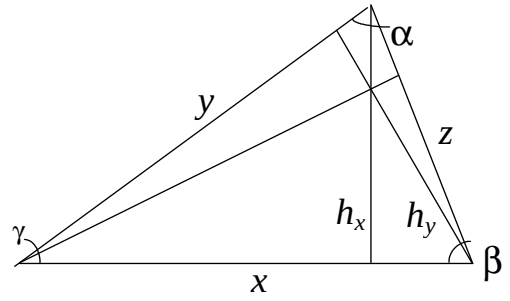
В дальнейшем будем считать известной обычную процедуру построения по терму t представляющего его дерева и введем конструкцию расширенного дерева так: каждый фрагмент на рис 7.6.а заменяется на более содержательный фрагмент на рис. 7.6.б., в котором показаны и операции и переменные терма.



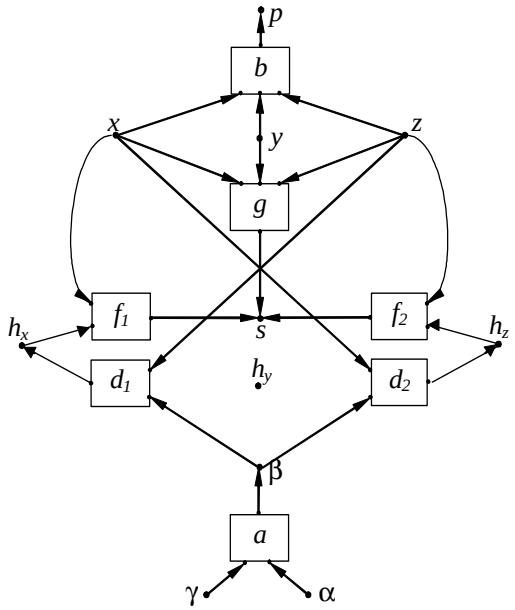
а) б)

Рис. 7.6

Большое число формально введенных определений полезно проиллюстрировать примерами. Прежде всего, посмотрим, как строится вычислительная модель.



а)



б)

Рис. 7.7.б

Пусть предметная область есть геометрия и необходимо описать понятие треугольник (рис.7.7.а). Напомним, что ключевая идея вычислительных моделей состоит в накоплении

множества различных алгоритмов предметной области и их взаимосвязей. Основными величинами, характеризующими треугольник, являются (рис. 7.7.а): x, y, z - стороны; p - полупериметр; α, β, γ - углы; h_x, h_y, h_z - высоты треугольника, опущенные на стороны x, y, z; s - площадь. Треугольни характеризуют и могие другие переменные и их взаимосвязи, но на текущий момент они не используются и не включаются в ПВМ.

Проанализируем ряд основных закономерностей, связывающих эти величины. Из формулы $\alpha+\beta+\gamma=180^\circ$, разрешая один из аргументов относительно двух других и вводя обозначение $a(X,Y)=180^\circ-X-Y$, получим

$$\alpha=a(\beta,\gamma),\ \beta=a(\alpha,\gamma),\ \gamma=a(\alpha,\beta).$$

Аналогично, используя формулу $p=(x+y+z)/2$, имеем

$$p=(x+y+z)/2=b(x,y,z);$$
$$x=2p-y-z=c(p,y,z);$$
$$y=2p-x-z=c(p,x,z);$$
$$z=2p-x-y=c(p,x,y),$$

где $b(X,Y,Z)$ обозначает функцию $(X+Y+Z)/2$, а $c(X,Y,Z)$ - функцию $2X-Y-Z$.

Выражение высот через стороны приводит к следующим соотношениям:

$$h_x=y\cdot\sin\gamma=d(y,\gamma)=z\cdot\sin\beta=d(z,\beta);$$
$$h_y=x\cdot\sin\gamma=d(x,\gamma)=z\cdot\sin\alpha=d(z,\alpha);$$

$$h_z = x \cdot \sin\beta = d(x,\beta) = y \cdot \sin\alpha = d(y,\alpha).$$

К ним следует добавить все обратные соотношения вида $y = h_x/\sin\gamma$; $\gamma = \arcsin(h_x/y)$ и т. д.

Наконец, площадь может быть получена по формулам

$$s = xh_x/2 = f(x,h_x) = yh_y/2 = f(y,h_y) = zh_z/2 = f(z,h_z) =$$
$$\sqrt{p(p-x)(p-y)(p-z)} = g(x,y,z).$$

(В последнем обозначении переменная p исключена, так как ее можно выразить через x, y, z). Конечно, выписанный список закономерностей, как и используемых параметров треугольника, не является полным, но он достаточен для иллюстративных целей.

Приведенные соотношения показывают принципиальную возможность вычисления одних параметров треугольника через другие. Для реализации этой возможности необходимо написать и включить в библиотеку программ соответствующие программные модули. Для нашего примера предположим, что на каком-то этапе эксплуатации в библиотеку включены модули для вычислений по следующим формулам:

$$p = b(x,y,z);\ s = g(x,y,z);\ s = f(x,h_x);$$
$$s = f(z,h_z);\ \ h_x = d(z,\beta);$$
(7.1)
$$h_z = d(x,\beta);\ \beta = a(\gamma,\alpha).$$

ПВМ, описывающая такое состояние системы, изображена на рис.7.7.б. Заметим, что в списках модулей (7.1) отсутствуют

отношения $\gamma=a(\alpha,\beta)$, $\alpha=a(\beta,\gamma)$, а также аналогичные соотношения для модулей *f* и *d*, хотя другие соотношения для этих модулей включены в (7.1), а, следовательно, эти модули должны быть программно реализованы. Кроме того, на рис.7.7.б имена модулей *f* и *d* имеют индексы. Все это является следствием определения ПВМ, принятого в этой главе. Для технических удобств простая модель определена так, что все операции имеют различные имена. Таким образом, операции для вычисления $s=f(x,h_x)$, $s=f(y,h_y)$, $s=f(z,h_z)$ считаются различными, несмотря на то, что отличаются лишь входными переменными.

Теперь можно попробовать сформулировать и решить задачу на этой модели. Пусть есть возможность измерить две стороны треугольника *x*, *z* и два противолежащих им угла α и β. Зная эти величины, требуется найти площадь фигуры. Это стандартная постановка задачи на вычислительной модели:

- известны значения переменных из некоторого подмножества *V*, здесь $V=\{x,z,\gamma,\alpha\}$;
- требуется найти значения переменных из множества *W*, здесь $W=\{s\}$.

Анализируя структуру связей модели, видим, что с помощью модуля *a* можно вычислить из γ и α переменную β; модули d_1 и d_2 позволяют вычислить переменные h_z и h_x из переменных *x*, *z*, β; а *s* можно найти либо из h_x и *x*, либо из *z* и h_z с помощью модулей f_1

и f_2. Таким образом, алгоритм вычисления s можно представить термами

$$t_1 = f_1(x, d_1(z, a(\gamma,\alpha))); \; t_2 = f_2(z, d_2(x, a(\gamma,\alpha)));$$

Значения определенных атрибутов для терма $t_1 = f_1(x, d_1(z, a(\gamma,\alpha)))$ (рис.7.8.)

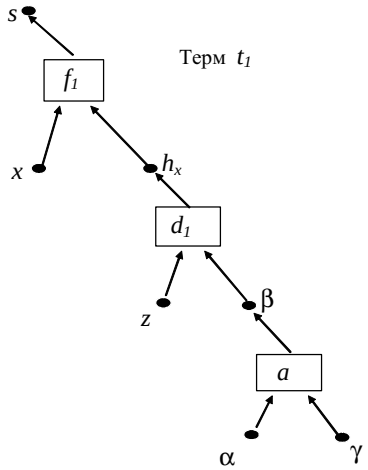


Рис.7.8

следующие:

$in(t_1)=\{x, z, \gamma, \alpha\}$, $out(t_1)=\{s\}$, $var(t_1)=\{x, z, s, \gamma, \alpha, \beta, h_x\}$

$top(t_1)=f_1$; $oper(t_1)=\{f, d, a\}$; $d(t_1)=3$;

$sbt(t_1)=\{t_1, x, d_1(z,a(\gamma,\alpha)), z, a(\gamma,\alpha), \gamma, \alpha\}$

Подтерм $t'=d_1(z,a(\gamma,\alpha)) \in t_1$, $\mathrm{out}_{t1}(t')=h_x$. Подтерм t' вычисляет переменную h_x в терме t_1, терм t_1 вычисляет переменную s. Это значит, что переменная h_x входит в $out(t_1)$ и при соответствующей интерпретации значение переменной может быть вычислено при

реализации терма. Операция d_1 используется в терме (операция входит в терм) t_1 для вычисления переменной h_x. Будем также говорить, что переменная h_x входить в терм t_1 (либо используется в терме t_1).

В процессе разработки модели на рис.7.7.а было известно, что она строится для конкретной предметной области - геометрии. Смысл всех объектов модели - переменных и операций - понятен (они интерпретированы) для нас. Закономерности геометрии нашли свое отражение, во-первых, в синтаксисе модели, т.е. в характере связей между операциями и переменными, во-вторых, в семантике, т.е. в специфике этих операций.

Когда модель построена и надо найти цепочку операций для вычисления одной группы переменных из другой, тогда ее семантические особенности уже не важны. В процессе планирования - построения цепочки - можно учитывать только синтаксис модели, абстрагируясь от специфики операций. В общем случае, одна и та же вычислительная модель может соответствовать различным предметным областям и только сопоставление операциям различных конкретных функций (процедур) определят ее специфику. Таким образом, вычислительная модель является неинтерпретированным объектом. Именно абстрагирование от конкретной интерпретации позволяет с помощью лишь структурной

составляющей создавать универсальные алгоритмы синтеза программ с реальными оценками сложности [6].

Однако для точной формулировки результатов без понятия интерпретации не обойтись. Возможны разные определения понятия интерпретации. Здесь выбрано то, которое включает фиксацию значений входных переменных алгоритма (переменных из V) и интерпретацию всех объектов, используемых для вычисления W из V.

Пусть $V \subseteq \mathbf{X}$ – некоторое подмножество переменных, играющее роль входных переменных алгоритма. *Интерпретация* $\boldsymbol{I}$ в области интерпретации D есть функция, которая сопоставляет:

- каждой переменной $x \in V$ - элемент $d_x = \boldsymbol{I}(x) \in D$, d_x называется значением переменной x в интерпретации $\boldsymbol{I}$ (далее просто значение);
- каждой операции $a \in \mathbf{F}$ - вычислимую функцию f_a: $D^m \rightarrow D^n$, $in(a)=\{x_1, x_2, ..., x_m\}$, $out(a)=\{y_1, y_2, ..., y_n\}$;
- каждому терму $t=a(t_1,t_2,...,t_m)$ - суперпозиция функций в соответствии с правилом $\boldsymbol{I}(a(t_1,t_2,...,t_m))=f_a(\boldsymbol{I}(t_1),\boldsymbol{I}(t_2),...,\boldsymbol{I}(t_m))$.

Пусть $t=a(t_1,t_2,...,t_m)$ - произвольный терм такой, что $in(a)=\{x1, x2, ..., xm\}$, $out(a)=\{y1, y2, ..., yn\}$. Тогда набору переменных $out(a)$ сопоставляется в интерпретации $\boldsymbol{I}$ набор (кортеж) значений

$$val(t)=(d_1,d_2,...,d_n)=f_a(val_{x1}(t_1),val_{x2}(t_2),...,val_{xn}(t_n))$$

Здесь $val_z(t)$ обозначает значение того компонента набора $val(t)$, который соответствует переменной z.

Впредь всегда предполагается, что каждой функции $f_a=\boldsymbol{I}(a)$ соответствует модуль mod_a, который и будет использоваться в программе для вычисления f_a. Если существуют два различных алгоритма вычисления переменной y (существуют два различных терма t_1 и t_2 таких, что $y\in out(t_1)\cap out(t_2)$, $in(t_1)\cup in(t_2)\subseteq V$) из одних и тех же начальных переменных V, то по смыслу модели эти два значения переменной y должны быть эквивалентны (равны в простейшем случае). Например, площадь треугольника может быть вычислена различными алгоритмами (различными термами), однако вычисленные значения должны быть равны (если, конечно, начальные данные действительно определяют треугольник и запрограммированные модули верно вычисляют функции). Это одна из причин, по которой следует строить избыточные вычислительные модели, в которых существует несколько алгоритмов вычисления переменных.

Дальше будем работать только с такими *корректными* интерпретациями. По определению, в корректной интерпретации для любой переменной y и любых термов t_1 и t_2 таких, что $y\in out(t_1)\cap out(t_2)$, значения $val_y(t_1)=val_y(t_2)$, т.е. оба терма *вырабатывают* одно и то же значение переменной y. Это позволяет говорить о значениях переменных, вычисляемых не

только отдельными термами, но и множествами термов. Это всегда далее имеется в виду.

Все термы из множества $T(V,F)$ обладают свойством: $t \in T(V,F) \rightarrow in(t) \subseteq V$. Термы множества $T(V,F)$ определяют все вычисления (все алгоритмы), которые могут быть произведены из переменных множества V (если заданы их значения).

Если ВМ циклическая, то множество термов $T(V,F)$ будет бесконечным и избыточным вот в каком смысле. Для циклической *BM* на рис.7.5, если множество $V=\{x\}$, то множество $T(V,F)$ в соответствии с его определением содержит счетное подмножество термов, показанное на рис.7.9. Термы t_1 и t_2 безусловно полезны, t_2 содержат новый способ вычисления x с использованием операции b. Терм t_3 уже избыточен, он содержит двукратное вычисление переменной x одной и той же операцией b, что не дает ничего нового. Такие термы называются *избыточными*, и впредь мы с ними работать не будем.

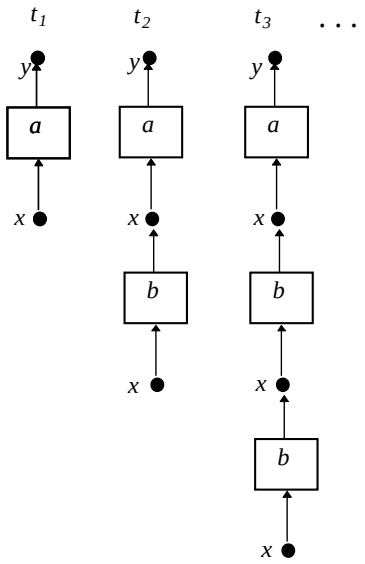


Рис. 7.9.

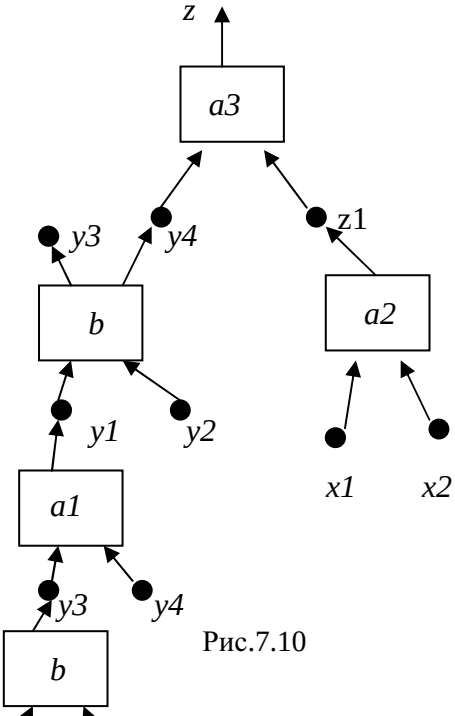


Рис.7.10

Определение. Терм называется *избыточным*, если на пути от входных переменных к верши ветви расширенного дерева, представляющего терм) одна и та же операция встречается более одного раза для вычисления одной и той же своей выходной переменной.

В примере на рис.5.9 операция *b* встретилась дважды в терме t_3 на пути от входной переменной *x* до выходной

переменной *y* для вычисления переменной *x*. Следовательно, терм t_3 (и все последующие термы t_4, t_5, ... на рис.7.9) избыточные. С другой стороны, терм на рис.7.10 не избыточный, в нем операция *b* встретилась дважды на ветви дерева, но использовалась для вычисления разных своих выходных переменных: сначала *y3*, а затем *y4*.

Множество всех не избыточных термов из $T(V,F)$ обозначается *T1*, $T1 \subseteq T(V,F)$. Впредь будем работать только с термами из *T1*. Ясно, что множество не избыточных термов *T1* может быть только конечным.

Определим теперь множество термов $T_V^W = \{t \in T1 \mid out(t) \cap W \neq \varnothing\}$. Это множество задает все вычисления, которые основаны на *V* и завершаются в *W*.

Определение. Множество термов $R \subseteq T_V^W$ такое, что $\forall x \in W \exists t \in R(x \in out(t))$ называется (V,W)-планом вычислений. Ясно, что (V,W)-план задает детерминант вычислимой функции, которая вычисляет переменные W из переменных *V*.

7.1.3.Оптимизация при планировании вычислений

Пусть $V \cap W = \varnothing$ и $R \subseteq T1$ - некоторый (V,W)-план. Это множество строится неоднозначно, и, следовательно, возможно его целенаправленное сокращение. Рассмотрим три критерия оптимизации сокращения:

1.минимум числа термов во множестве;

2.минимальное число переменных, от которых зависят термы множества;

3.наименьшая максимальная глубина совокупности термов.

Рассмотрим вначале принцип оптимизации для первого критерия.

Все множество термов R разбивается на классы R_a, $a \in F$, так, что в R_a входят термы вида $a(t1,...,tm)$. Ввиду корректности интерпретации понятно, что любой терм $t \in R_a$ может быть взят для вычисления некоторой переменной из $out(a)$ и $\{t\} \subseteq T1$. Таким образом, на первом этапе сокращения множества R можно оставить по одному представителю из каждого класса R_a.

Пусть $A \subseteq F$ - множество всех операций, завершающих термы “сокращенного” описанным способом множества $P \subseteq R$. Теперь задача минимизации числа термов сводится к одному варианту комбинаторной задачи о наименьшем покрытии. Дано множество W и некоторое множество подмножеств W_a:$\{out(a):a \in A\}$, покрывающих в совокупности W. Требуется найти наименьшее число подмножеств, в объединении покрывающих все множество W. Задача может быть решена, например, с помощью следующего алгоритма.

Процедура 1:

1.Вводим для каждой $x \in W$ дизъюнкцию переменных $v_a \vee v_b \vee \ldots \vee v_c$, где $\{a, b, ..., c\}$ - список всех операций, вырабатывающих x. Эта дизъюнкция выражает возможность вычисления переменной x любым термом t из $T1$, $\mathrm{top}(t) \in \{a, b, ..., c\}$.

2.Берем конъюнкцию полученных дизъюнкций. Образованная конъюнктивная форма выражает “высказывание” о том, что для покрытия W нужно покрыть каждую из переменных, которая, в свою очередь, может быть покрыта одним из v_a.

3.Преобразуем полученную форму в дизъюнктивную. Произведя эквивалентные преобразования, получим все альтернативы покрытия W.

4.Выбираем кратчайшую по длине альтернативу. Соответствующий набор термов и есть искомый план.

Пусть, например, $A=\{a, b, c, d\}$, $W=\{x, y, z, v\}$, $out(a)=(x, y)$, $out(b)=(z)$, $out(c)=(z, v)$, $out(d)=(y, z)$. Тогда соответствующая конъюнктивная форма $v_a \wedge (v_a \vee v_d) \wedge (v_b \vee v_c \vee v_d) \wedge v_c$ после приведения превращается в

$$(v_a \wedge v_b \wedge v_c) \vee (v_a \wedge v_b \wedge v_c \wedge v_d) \vee (v_a \wedge v_c) \vee (v_a \wedge v_c \wedge v_d)$$

Самый короткий из дизъюнктивных членов $v_a \wedge v_c$; таким образом, $\{a(...), c(...)\}$ представляет собой наименьший по численности термов план.

Второй вид оптимизации - минимизация множества «входных» переменных - можно использовать для уменьшения числа медленных операций ввода или обращения к базам данных. Решение задачи оптимизации дает

Процедура 2:

1. Для каждой $x \in W$ выписываем выражение

$(v_{11}^{t_1} \wedge ... \wedge v_{1m_1}^{t_1}) \vee ... \vee (v_{k1}^{t_k} \wedge ... \wedge v_{km_k}^{t_k})$,

где $v_{i1}^{t_i}, \wedge ... \wedge v_{im_i}^{t_i}$ - конъюнкция входных переменных терма t_i;

$t_1, ..., t_k$ - список всех термов, вырабатывающих x.

2. Берем конъюнкцию полученных выражений по всем $x \in W$.

3. Приводим эту конъюнкцию к дизъюнктивной нормальной форме. Полученная форма представляет все наборы исходных переменных и ассоциированные с ними наборы термов, «вырабатывающие» в результате множество W.

4. Выбираем кратчайшую альтернативу полученной формы: выражение, содержащее наименьшее число переменных. Множество термов, участвующее в ней в качестве верхних индексов, представляет собой искомый план.

Третий тип оптимизации - выбор плана с наименьшей глубиной самого глубокого терма. При организации параллельного вычисления аргументов подобный план имеет наименьшее время реализации.

Пусть V, $W \subseteq \mathbf{X}$. Рассмотрим процедуру формирования (V,W)-плана.

Процедура 3:

1. Вводим для каждой переменной $x \in \mathbf{X}$ пару вспомогательных переменных: n_x - минимальный номер шага, на котором x может быть вычислена из V; a_x - операция, с помощью

которой она вычисляется. Для $x \in V$ полагаем $n_x := 0$, $a_x := x$. Вводим также вспомогательное множество операций P, и первоначально полагаем $P = \{a_x : x \in V\}$.

2. Если $y \notin V$ и A - множество операций $a \in F$ таких, что $in(a) \subseteq \bigcup_{b \in P} out(b)$, $y \in out(a)$, то полагаем $n_x = min\{min\{max\{n_x: x \in in(a)\}\}+1, n_y\}$, а в качестве a_y берем то значение, на котором достигается минимум, или прежнее значение a_y, если минимальным значением является прежнее значение n_y. Все такие a_y включаем в множество P.

3. Если никакое выполнение шага 2 не меняет значений n_y, то строим искомый план путем «наращивания» термов от вершин к аргументам:

а) для каждого $v \in W$ берем a_v в качестве вершины формируемого терма, вычисляющего v. Если a_v не есть $x \in V$, то включаем эту вершину в специально выделенное множество «растущих» вершин P_t формируемого терма;

б) пусть часть терма построена, и P_t - множество его растущих вершин. Тогда для каждой $a_v \in P_t$ вхождение a в формирующийся терм заменяется на вхождение $a(a_{x_1},...,a_{x_m})$, где $(x_1,...,x_m) = in(a)$;

в) процесс построения плана заканчивается, если для всех формируемых термов множество «растущих» вершин пусто.

Таким образом, процедура 3 вначале проводит разметку модели, наподобие того, как это делается при построении критического пути, затем выбирает те термы, которые «лежат» на критических (минимальных) путях.

Итак, если $(R \subseteq T_V^W) \& (W \subseteq out(R))$ - некоторое конечное множество термов, то

а) процедура 1, примененная к R, дает в результате совокупность (V,W)-планов, имеющих наименьшее число термов

из всех (V,W)-планов, включающихся в R;

б) процедура 2, примененная к R, дает в результате совокупность (V,W)-планов P таких, что для любого другого плана $L \subseteq R$, вычисляющего все W, выполняется

$$\left| \bigcup_{t \in L} \text{in}\ (t) \right| > \left| \bigcup_{t \in P} \text{in}\ (t) \right|;$$

в) процедура 3, примененная к модели с выделенными множествами V, W, дает в результате (V,W)-план P такой, что для любого другого (V,W)-плана L, вычисляющего все W, выполняется

$max\{d(t): t \in L\} \geq max\{d(t): t \in P\}$

7.1.4.Генерация параллельных программ

После того как построен оптимальный по одному из критериев план *P*, требуется составить реализующую его программу (более точно, схему программы, так как интерпретация не задана). Рассмотрим три типа таких программ, предписывающих параллельную обработку данных в соответствии с порядком, определяемым термами плана *P*.

Первый тип - самая простая программа - представляет собой совокупность параллельных присваиваний переменным $v_i \in W$ вырабатывающих их термов плана *P*.

$$\begin{cases} v_1 := t^1; \\ \quad \vdots \\ v_k := t^k; \quad v_i \in W, \quad t^i \in P, i = 1,\ldots,k \end{cases}$$

Построение такой программы возможно в случае, если каждая операция, участвующая в термах t^i, имеет ровно одну выходную переменную и после срабатывания “обладает значением этой переменной”. Само собой разумеется, что запись (7.2) должна быть синтаксически правильна для языка, в котором записана программа.

Например, для плана $P=\{a(t,g(x,t)),\ d(x,t,g(x,t))\}$ реализующая его программа типа (7.2) будет иметь вид

$$\begin{cases} y := a(t, g(x,t)); \\ v := d(x,t,g(x,t)); \end{cases}$$

Второй тип программы - последовательно-параллельное задание вычисления с помощью операторов ветвления и слияния: **fork**, **join**. Считаем, что пара таких операторов задает автономное параллельное вычисление участков (ветвей), лежащих между **fork** и **join** и разделенных запятой. Таким образом, все переменные локализованы в своих ветвях. Этот тип программ строится для операций общего вида, без ограничений на множество выходных переменных.

Пусть $P=\{t^1,\ t^2,\ ...,\ t^k\}$. Тогда синтезированная программа имеет вид

$$S(P)\begin{cases} S(t^1), \textit{если} & k=1; \\ \textbf{for k}\ \ S(t^1), S(t^2), \ldots S(t^k)\textbf{join}, \textit{если} & k\neq 1, \end{cases}$$

где $S(t^i)$ - параллельная программа для вычисления терма $t^i = a\left(t_1, \ldots, t_m\right)$ - в свою очередь, имеет вид

$$S(t^i) = \begin{cases} a, & \textit{если} \quad m=0; \\ t_1; a, & \textit{если} \quad m=1; \\ \textbf{for k}\ \ S(t_1), \ldots, S(t_m)\textbf{join}; a, & \textit{если} \quad m\rangle 1. \end{cases}$$

В этих обозначениях a выполняется после выполнения всех ветвей $S(t^i)$, которые вычисляют входные переменные операции a. Для завершенности предполагаем, что для $x\in V$, $S(x)=$*ввод*(x), где "*ввод*" - оператор, генерирующий значение переменной x.

Программа $S(P)$ имеет структуру, подобную структуре терма. Так, для рассмотренного примера - плана P - имеем программу:

$S(P)$=**fork** $S(a(t,g(x,t)))$, $S(d(x,t,g(x,t)))$ **join**=

fork **fork** *ввод*(t), **fork** *ввод*(x), *ввод*(t)**join**; g **join**; a, **fork** *ввод*(x),

ввод(t) **fork** *ввод*(x), *ввод*(t)**join**; g **join**; g **join**.

Третий тип - в качестве объектной программы взята асинхронная программа (A-схема), задающая вычисление с помощью спусковых функций. A-схема S, реализующая (V,W)-план P, строится следующим образом. Пусть $\overline{P}$ - множество всех подтермов множества P, не являющихся переменными из V. Введем для каждого $t \in \overline{P} \setminus P$ и $x \in \mathrm{out}(t)$ ячейку информационной памяти x^t, а в информационную память для $x \in V \cup W$ - ячейку с тем же обозначением x. В качестве множества операций возьмем множество всех вхождений операций, "завершающих" термы множества $\overline{P}$, a^t обозначает тот экземпляр операции a, который "завершает" терм $t \in \overline{P}$. Если $t=a(t_1,...,t_m)$, $in(a)=(x_1,...,x_m)$, $out(a)=(y_1,...,y_n)$, то $in(a^t)=(x_1^{t_1},...,x_m^{t_m})$. Если $t \in P$, то $out(a^t)=out(t)$, если $t \notin P$, то $out(a^t)=(y_1^t,\ldots,y_n^t)$. В управляющую память для каждого a^t введем управляющую переменную $\overline{a}^t$, для каждого

$x \in V$ - управляющую переменную $\overline{x}$. Первоначально $\forall\, t(\overline{a}^t = 0)$, $\forall x \in V(\overline{x} = 1)$; после ввода x, $\overline{x}$ обращается в нуль. Спусковая функция $trf(a^t)$ для $t=a(t_1, ..., t_m)$ определяется так:

$$trf(a^t)=(\overline{a}^{t_1}=1)\wedge \ldots \wedge(\overline{a}^{t_m}=1).$$

Управляющая функция $c(a^t)$ после срабатывания a^t устанавливает значение переменной $\overline{a}^t$, равное единице. Этими правилами схема S полностью определена. Схема S строилась таким образом, чтобы “имитировать” вычисление термов плана P: вхождение a^t в терм t “срабатывает”, как только для него вычислены аргументы, а в ячейку $\overline{a}^t$ заносится признак срабатывания. “Расклейка” ячеек по числу подтермов исключает конкуренцию над памятью и обеспечивает реализацию схемой S в точности множества термов P.

Для примера плана $P=\{a(t,g(x,t)),d(x,t,g(x,t))\}$: множество операций включает a, g, d, информационные переменные - t, x, i^j, y, v, управляющие переменные - $\overline{t}$, $\overline{x}$, $\overline{g}^{g(x,t)}$ (далее обозначаемая просто $\overline{g}$), начальные значения - $\overline{t}=1$, $\overline{x}=1$, $\overline{g}=0$. Спусковые функции:

$$tr(g)=(\overline{t}=1)\wedge(\overline{x}=1);\ tr(a)=(\overline{t}=1)\wedge(\overline{g}=1);$$

$$tr(d)=(\overline{x}=1)\wedge(\overline{t}=1)\wedge(\overline{g}=1).$$

7.2.Алгоритмы синтеза параллельных программ

Итак, в первом параграфе главы рассмотрены основные понятия метода синтеза параллельных программ на вычислительных моделях. Этого, конечно, недостаточно для какой-либо реализации метода, самой характерной и общей из которых является система синтеза параллельных программ (ССПП). Описанию базовых алгоритмов, необходимых для создания такой ССПП, посвящен этот параграф.

7.2.1.Общая схема синтеза параллельной программы

Общая схема ССПП приведена на рис. 7.11. На ней показаны основные этапы на пути от формулировки задачи к программе ее решения. При реализации ССПП эта схема может, естественно, модифицироваться.

Рис.7.11.

На первом предварительном этапе составляется описание ПО в виде множества понятий ПО. Каждое понятие ПО определяется ПВМ и может запоминаться в библиотеке моделей и в дальнейшем использоваться для формулировки задачи. Такое описание делается специалистом в этой конкретной ПО, его знания о ПО запоминаются, таким образом, в ССПП в форме ПВМ. Формулировка задачи представляет собой ПВМ, при конструировании которой могли быть использованы понятия из библиотеки моделей, и оператор постановки задачи, определяющий множества входных *V* и выходных *W* переменных задачи.

Рассмотрим пример описания ПО *геометрия* и формулировки

задачи. Прежде всего описывается ПО *геометрия.*

Пусть *геом:* (*треугольник*: *(x, y,* z, *a, s:* **real**;

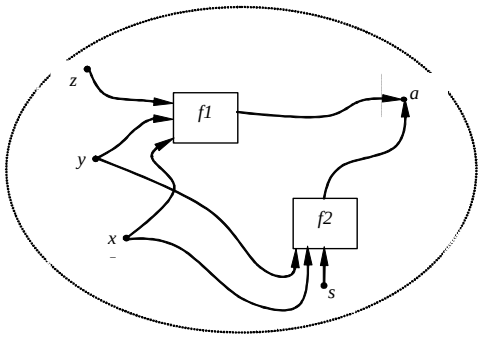
операция f_1 **in** *x, y, z* **out** *a*;

операция f_2 **in** *x, y,* s **out** *a*;);

окружность: *(r, s:* **real;**

операция f_3 **in** *r* **out** *s*;););

Это описание (очень неполное) транслируется и заносится в библиотеку моделей *геом.* В ней появляются ПВМ (понятие) *треугольник* (рис. 7.12), в которой геометрический объект *треугольник* характеризуется длинами трех сторон *x, y, z,* величиной угла *a* и площадью *s* (представлены в ПВМ одноименными переменными), а также операциями f_1 и f_2 для вычисления угла *a.*



Треугольник

Рис. 7.12.

Понятие *окружность* (рис. 7.13) определяется радиусом *r* и площадью *s,* операция f_3 вычисляет площадь *s* из радиуса *r.*

Понятно, что степень детальности описания каждой ПО зависит от задач, которые необходимо решать.

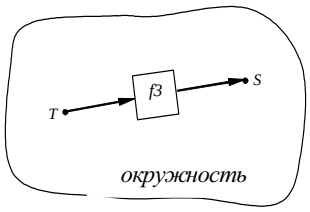


Рис. 7.13.

Данное описание ПО *геом* позволяет сформулировать такую задачу: даны треугольник *tr* с известными длинами двух сторон и окружность *k* с радиусом *r*. Требуется определить угол *a* в треугольнике *tr*, если площади *tr* и *k* равны. Условие задачи описывается в ССПП как ПВМ:

Пусть *задача: (tr : треугольник x=x0, y=y0;*

k: окружность r=r0, s=tr.s;);

на *задача* **вычислить** *tr.a* **из** *k.r, tr.x, tr.y;.*

Графическое представление ПВМ *задача* показано на рис. 7.14, состоит она из двух понятий: *tr* и *k*. Понятие *tr* строится с помощью понятия *треугольник*. Отличие состоит в том, что понятие *треугольник* определяет множество всех треугольников, а понятие *tr* - множество всех треугольников, длина стороны *x* у которых равна *x*0, а стороны *y* — *y*0 (в примере *x*0, *y*0 и *r*0 — вещественные константы). Понятие *k* определяет окружность радиуса *r*0 и, кроме того, конструкция *k.s=tr.s* показывает, что переменная *s* в понятии *tr* и переменная *s* в понятии *k* должны быть отождествлены, т.е., площади круга и треугольника в задаче равны в корректной интерпретации.

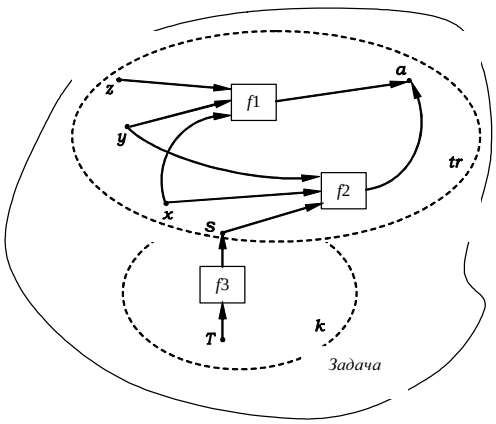


Рис. 7.14.

Оператор постановки задачи (конструкция **на** *задача* **вычислить**

tr.a **из** *k.r*, *tr.x*, *tr.y*) указывает, что дано (V={*k.r*, *tr.x*, *tr.y*}) и что требуется вычислить W={*tr.a*} на ПВМ *задача.*

Формулировка задачи (т.е. ПВМ и оператор постановки задачи) поступает на вход транслятора. При трансляции решаются следующие задачи:

- проверка синтаксической правильности формулировки задачи;
- формирование описания ПВМ в том внутреннем представлении, которое необходимо планировщику.

Первая задача традиционна, а вторая заключается в преобразовании ПВМ из текстового представления во внутреннее, табличное. Необходимые для решения этих задач алгоритмы достаточно очевидны.

Внутреннее представление ПВМ **C**=(**X**,**F**) состоит из двух таблиц: *TX* и ОП. Каждая строка таблицы *TX* имеет вид (x,$A(x)$,$comp(x)$), а таблицы ОП - (a,$in(a)$,$out(a)$), где $x \in \mathbf{X}$, $a \in \mathbf{F}$, $A(x)=\{a \in \mathbf{F} | x \in in(a)\}$, $comp(x)=\{a \in \mathbf{F} | x \in out(a)\}$.

Таким образом, каждая строка *TX* содержит описание всех дуг, входящих и выходящих в/из переменной x (в графическом представлении **C**), а каждая строка ОП — описание всех дуг входящих и выходящих в/из операции a. Структура понятий ПВМ в таблицах не сохраняется.

Задача планирования заключается в построении множества T_V^W не избыточных термов. Конечность множества T_V^W

очевидна, однако число его элементов может оказаться очень большим, что иллюстрируется следующим примером:

пример 1: (*z*, *y*, x_1, ..., x_m**: real;**

операция *a* ***in*** x_1, …, x_m **out** *z*;

операция b_1 **in** *y* **out** x_1, ..., x_m;

. . . . . .

операция b_n ***in*** *y* **out** x_1, ..., x_m;);.

Здесь *m* и *n* — натуральные числа, многоточия понимаются обычным образом. Если *V*={*y*}, *W*={*z*}, то T_V^W содержит n^m термов. Понятно, что построение T_V^W быстро становится нецелесообразным при росте *n* и *m*. Поэтому следует строить и хранить описание T_V^W, имеющее линейную сложность, а затем на его основе конструировать необходимые термы. Далее, в параграфе 7.2.2, описан алгоритм планирования, который удаляет из таблиц *TX* и ОП все те переменные и операции, которые не используются ни в каком терме из T_V^W, а также приведен алгоритм выбора, предназначенный для построения (*V*,*W*)-плана. Таким образом, после планирования в *TX* и ОП остается описание ПВМ, каждая переменная и операция которой используются хотя бы в одном терме из T_V^W.

На последних двух этапах конструируется ПП решения

сформулированной задачи. Конструирование ПП заключается в построении (V,W)-плана T и компиляции (записи в некотором языке программирования) параллельной программы, реализующей T. Множество термов T_V^W определяет множество TVW всех (V,W)-планов, каждому (V,W)-плану $R \in TVW$ соответствует множество ПТ всех тех ПП, которые реализуют T.

На рис. 7.15 показана ПВМ, если $V=\{z_1,z_2\}$, а $W=\{x\}$, то множество $T_V^W=\{a(b(z_1),\ b(z_1)),\ a(b(z_1),\ c(z_2))\}$, множество $TVW=\{a(b(z_1),\ b(z_1))\},\ \{a(b(z_1),\ c(z_2))\}\}$, а ($V$,$W$)-план $\{a(b(z_1),\ c(z_2))\}$ реализует каждая из трех следующих программ:

1) **begin** $y_1:=b(z_1)$; $y_2:=c(z_2)$; $x:=a(y_1,y_2)$ **end;**

2) **begin** $y_2:=c(z_2)$; $y_1:=b(z_1)$; $x:=a(y_1,y_2)$ **end;**

3) **begin fork** $y_1:=b(z_1)$; $y_2:=c(z_2)$ **join;** $x:=a(y_1,y_2)$ **end;.**

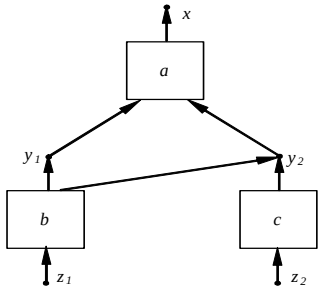


Рис. 7.15

Здесь и везде далее имеется в виду, что операции a при интерпретации сопоставляется функция $f_a=I(a)$, которая может

быть вычислена в ПП модулем mod_a. Обращение к модулю в разных языках программирования делается по-разному, поэтому при задании ПП для единообразия используется запись вида $y_1=a(z_1)$, которая обозначает обращение к mod_a с входной переменной z_1, для вычисления значения выходной переменной y_1, значения других выходных переменных mod_a не используются. Аналогично, выражение "...для выполнения операции a в мультикомпьютере есть необходимые ресурсы..." означает, что для выполнения модуля mod_a в мультикомпьютере есть ресурсы.

Часто оказывается, что функция f_a вычисляется разными алгоритмами с разной эффективностью в различных подобластях своей области определения. Для повышения эффективности ПП полезно было бы в этом случае использовать для вычисления f_a в разных подобластях разные модули. Делать это можно с помощью условных операций.

Условной операции a при интерпретации сопоставляется функция $f_a=I(a)$, которая может быть вычислена условной процедурой вида: **if** q_a **then** mod_a, модуль mod_a вычисляет функцию f_a в области истинности предиката q_a. Понятно, что если для вычисления функции f_a полезно использовать, к примеру, три разных модуля, то в ПВМ появляются три разные условные операции, которым сопоставляется при интерпретации одна и та же функция f_a, но вычисляется она разными условными

процедурами. Наличие в ПВМ условных операций должно учитываться, конечно же, при построении плана вычислений. Чтобы избежать двусмысленности там, где это может случиться, не условные операции будут называться *безусловными*.

7.2.2.Планирование алгоритма

Разработано много различных алгоритмов планирования. В параграфе рассматривается хорошо реализуемый алгоритм, который позволяет строить все термы из T_V^W и имеет линейную временную сложность относительно числа дуг в графическом представлении ПВМ. Понятия и обозначения, введенные при описании алгоритма, неоднократно понадобятся и в дальнейшем, так как на его основе строится еще ряд алгоритмов планирования на вычислительных моделях.

Пусть задана ПВМ $\mathbf{C}=(\mathbf{X},\mathbf{F})$, которая после трансляции представлена в виде двух таблиц *TX* и ОП. Каждая строка таблицы *TX* имеет вид $(x,A(x),comp(x))$, а таблицы ОП - $(a,in(a),out(a))$, $x\in\mathbf{X}$, $a\in\mathbf{F}$, $comp(x)=\{a\in\mathbf{F}\mid x\in out(a)\}$, $A(x)=\{a\in\mathbf{F}\mid x\in in(a)\}$.

Чтобы не делать далее несущественных оговорок, предполагается, что $in(a)\neq\varnothing$ для любых $a\in\mathbf{F}$. Алгоритм планирования состоит из двух частей: восходящей и нисходящей.

В *восходящей* части алгоритма строятся множества переменных и операций, используемых в термах из множества

$T_V=T(V,F)$.

Обозначим $V_0=V$, тогда

$$F_0=\{a\in\mathbf{F}\mid in(a)\subseteq V_0\}=\bigcup_{x\in V_0}\{a\in A(x)\mid in(a)\subseteq V_0\}$$

содержит все операции ПВМ такие, что $in(a)\subseteq V_0$. Далее формируется множество $V_1=\{x\in\mathbf{X}\mid x\in out(a)\wedge a\in F_0\}\cup V_0$, на основе V_1 строится множество

$$F_1=\bigcup_{x\in V_1\setminus V_0}\{a\in A(x)\mid in(a)\subseteq V_1\}$$

и т. д. до тех пор, пока при некотором целом положительном k не окажется, что $F_k=\varnothing$. На этом завершается восходящая часть алгоритма планирования. Множества V_i и F_i, $i=0,...,k$, содержат все переменные и операции, используемые в термах из множества T_V.

Если $W\not\subseteq V_k$, то планирование можно прекращать, так как в этом случае существует переменная в W, которая не вычисляется никаким термом из множества T_V, и, следовательно, не существует алгоритма решения сформулированной задачи на основе имеющихся знаний о ПО. В этом случае говорим, что сформулированная задача синтеза *неразрешима*. В противном случае можно начать строить множества переменных и операций, используемых в термах из T_V^W. Обозначим $F^*=\bigcup_{i=0}F_i$, и определим множества

$$G_i = \bigcup_{x \in H_{i-1}} \left\{ a \in F^* \middle| a \in \mathrm{comp}(x) \wedge a \notin \bigcup_{m=1}^{i-1} G_m \right\},\ H_i = \bigcup_{a \in G_i} \mathrm{in}(a).$$

Построение множеств G_I и H_i завершается, когда при некотором целом положительном r окажется $G_r = \varnothing$. Множества G_I и H_i, i = 1, ..., r, содержат все переменные и операции, используемые в термах из множества T_V^W . Кроме того, в них остаются операции и переменные, необходимые для построения некоторых дублирующих вычислений.

На рис. 7.16 показаны множества F_i и V_i, образовавшиеся в результате восходящей части алгоритма планирования на ПВМ (рис. 7.17) при V={ x_1, x_2 }, W={ x_{10} }, а на рис.7.18 - множества G_I и H_i, сформированные в нисходящей части алгоритма планирования. После завершения планирования в таблицах TX и ОП остаются лишь переменные и операции из множеств H_i и G_I, остальные удаляются (рис. 7.19).

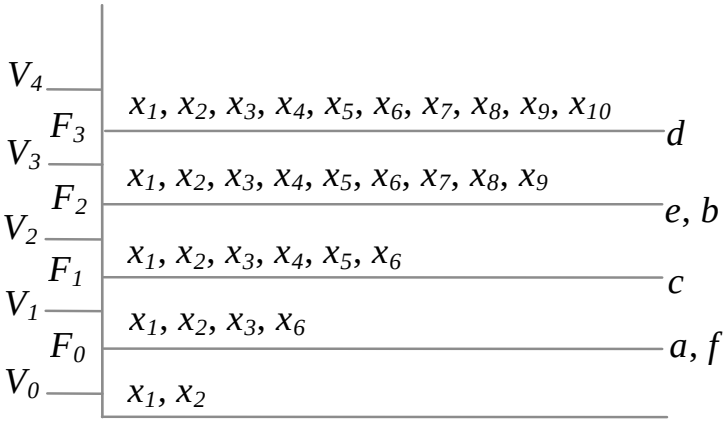


Рис.7.16.

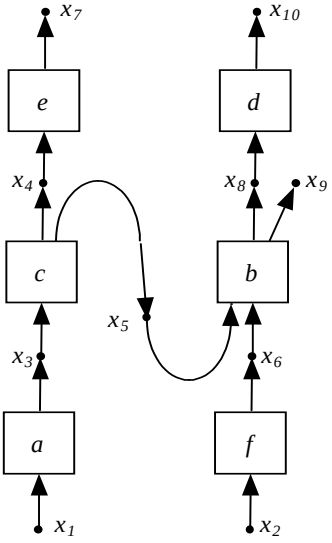


Рис.7.17.

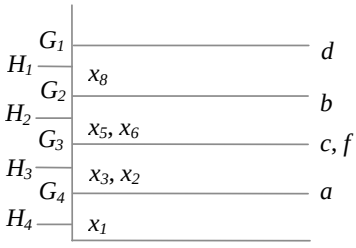


Рис.7.18.

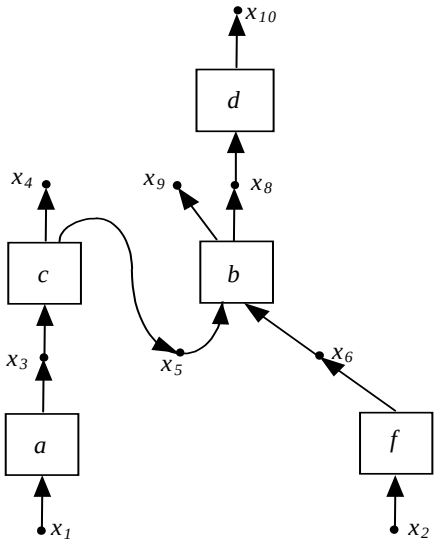


Рис.7.19.

Таким образом, результатом планирования является ПВМ, оставшаяся от **C** после удаления из *TX* и ОП “лишних” переменных и операций. Множество T_V^W не строится, подходящий в некотором смысле (*V*,*W*)-план *T* строится в каждом конкретном случае процедурой выбора алгоритма.

В случае, когда $W \not\subseteq V_k$, сформулированная задача синтеза оказывается неразрешимой и необходимо изменить формулировку задачи, т. е. либо уменьшить *W*, удалив из него невычислимые переменные, либо расширить *V*, включив в него

такие новые переменные, что станут вычислимыми все переменные из W. Для уменьшения затрат на расширение V может быть использован алгоритм планирования. Для этого необходимо выполнить его *нисходящую* часть из множества переменных $W'=W\backslash V_k$ с использованием всех операций из F. Все переменные из построенных при этом множеств H_i, $i=1, 2, ..., r$, являются кандидатами на включение в V. Из них человек может выбрать те переменные, значения которых ему доступны. Нетрудно также построить человеко-машинные алгоритмы, помогающие сделать такой выбор.

Из описания алгоритма следует, что проверка условия $\mathrm{in}(a)\subseteq V_i$ делается не более одного раза для каждой входной дуги произвольно взятой операции a, а проверка условия $out(a)\cap H_{i-1}\neq\varnothing$ - не более одного раза для каждой выходной дуги a. Понятно, что алгоритм планирования имеет линейную относительно числа дуг в графическом представлении ПВМ временную сложность, если в качестве элементарных шагов алгоритма взять проверки $\mathrm{in}(a)\subseteq V_i$ и $\mathrm{out}(a)\cap H_{i-1}\neq\varnothing$.

Алгоритм эффективно реализуется. При реализации алгоритма переменные и операции в *TX* и ОП могут кодироваться целыми положительными числами. Например, если в ходе трансляции в тексте встретилось j переменных, то очередная переменная получает номер $j+1$. Таким образом, n-я строка таблицы *TX* содержит описание n-й переменной, а m-*я* строка

таблицы ОП – описание m-й операции. Для представления всевозможных множеств —$A(x)$, $in(a)$, V_i, F_i и т. д., можно использовать битовые шкалы. Шкала V_i, к примеру, содержит в k-й позиции единицу, если переменная номер k принадлежит V_i. Применение битовых шкал сводит проверку условий $in(a) \subseteq V_i$ и $out(a) \cap H_{i-1} \neq \varnothing$ к двум логическим операциям.

Как можно было бы определить вычисления на вычислительной модели? Самый естественный способ следующий. Вначале вносятся значения во все переменные из V. Затем выполняются все операции из F_1 (выполняются модули mod_a для всех $a \in F_1$), при этом (в соответствии с определением интерпретации) вычисляются значения переменных из $V_1 \backslash V_0$. Далее выполняются все операции из F_2 и т.д. до F_k. Ясно, что такие вычисления могут быть весьма неэффективными, поскольку будут реализованы все алгоритмы вычисления W, которые определены моделью - и самые хорошие и самые плохие. Кроме того, могут быть вычислены и переменные, которые не входят в W и которые не надо вычислять вообще.

Основной смысл планирования именно в том и состоит, чтобы до начала вычислений статическим анализом

а)удалить операции и переменные, не участвующие в вычислении W,

и

б)подготовить информацию для конструирования приемлемого по качеству или даже оптимального алгоритма вычисления *W*.

Это, конечно, существенная и высокоэффективная оптимизирующая процедура. Известно, что наибольший оптимизационный эффект при построении решении задачи извлекается именно на этапе конструирования алгоритма. На всех остальных этапах эффект оптимизации значительно ниже.

7.2.3.Выбор алгоритма

Следующий после планирования этап - построение (V,W)-плана T. В параграфе 7.1. предложено несколько способов нахождения T, оптимального относительно довольно простых функционалов. Для более сложных функционалов следует использовать человеко-машинный алгоритм, с помощью которого человек сможет при необходимости управлять процессом построения T либо целенаправленно модифицировать ранее сконструированный (V,W)-план. Такая необходимость возникает в тех случаях, когда автоматически синтезированная программа $П_T$ не удовлетворяет человека по каким-либо причинам. Рассмотрим один такой алгоритм, который называется *алгоритмом выбора.*

1.Термы строятся из множества W, т. е. для каждой переменной $x \in W$ выбирается из множества $comp(x)$ одна из операций, которая вычисляет x, затем выбираются операции, которые вычисляют те входные переменные ранее выбранной операции, которые не входят в V и т.д., пока не будет построен терм t, вычисляющий x, и $in(t) \subseteq V$. Это случится обязательно в силу свойства TX и ОП (планировщик уже удалил из TX и ОП операции, которые не используются для вычисления переменных из W). При выборе операции необходимо, конечно, следить за тем, чтобы построенный терм принадлежал множеству T_V^W.

2. Для выбора операции могут применяться следующие

стратегии.

А). Выбирается произвольная операция, если ее использование в (V,W)-плане не запрещено человеком. Построенный по этой стратегии (V,W)-план T называется произвольным.

Б).Выбирается операция, для реализации которой в мультикомпьютере есть ресурсы. Здесь предполагается, что в ССПП может быть введена информация, описывающая как мультикомпьютер, так и свойства модулей, задающих интерпретацию операций ПВМ, к примеру, требуемый для выполнения модуля процессор. Возможен случай, когда сделать выбор операции не удается (ни одна операция из comp(x) не может быть выполнена на данном конкретном мультикомпьютере), и тогда нужно будет отменить сделанный ранее выбор операции и взять, если можно, другую. Чтобы избежать такой ситуации, исключение операций, для реализации которых в мультикомпьютере нет подходящих ресурсов, лучше делать на этапе трансляции (не включать их в TX и ОП).

В). Из всех операций, вычисляющих x, выбирается операция $a \in F_i$ с минимальным номером i, а если таких несколько — то любая. В соответствии с этой стратегией строятся термы минимальной глубины. Такой выбор можно сделать всегда.

Г). Выбирается операция из comp(x), которая может быть выполнена на том же процессоре p_i, что и некоторая другая

операция b, которая уже включена в конструируемый (V,W)-план и $x \in in(b)$. Если такой операции не окажется в $\mathrm{comp}(x)$, то в $comp(x)$ ищется операция, которая может быть выполнена на процессоре p_2, связанном с p_1 коммуникационным каналом, и т. д. Если выбор операции сделать не удастся, придется отменить ранее сделанный выбор операции и попытаться выбрать другую. Может случиться, что выбрать операцию в конечном итоге не удается, это означает, что в мультикомпьютере есть процессоры не связанные каналами коммутации с другими процессорами мультикомпьютера, и конструирование ПП с использованием всех ресурсов мультикомпьютера окажется невозможным.

Д). Если переменная x используется в одном из ранее построенных термов из T, в котором она вычисляется операцией a, то выбирается операция a, а если не используется, но в ранее построенных термах применяются операции $b_1, \ldots, b_n$ такие, что $x \in out(b_1)$, $i=1, 2, \ldots, n$, то выбирается любая из них. Когда ни одно условие не выполнено — берется произвольная операция из $comp(x)$. Применение этой стратегии приводит к тому, что переменная x всюду, где она используется в термах T, вычисляется одним и тем же подтермом, а в синтезированной ПП может уменьшиться общее количество модулей. Если при конструировании ПП нужно экономить ресурсы мультикомпьютера, то реализуется один только этот подтерм, а вычисленное значение переменной передается в те фрагменты

ПП, где оно должно быть использовано.

Е). Выбирается любая из операций, использование которых в (V,W)-плане рекомендовано пользователем, а если таких нет в comp(x), то произвольная.

Ж). Выбирается любая операция из тех, входной набор которых содержит наибольшее число переменных, рекомендованных пользователем для использования в (V,W)-плане .

3. Алгоритм завершает работу, когда построен (V,W)-план T, либо выбор операции сделать не удается.

Следующие два замечания дополняют описание этого алгоритма. В процессе построения терма t при выборе операции, вычисляющей переменную x, алгоритм всегда пытается взять безусловную операцию. Если взять безусловную операцию невозможно, но есть условные операции b_1, ..., b_n, вычисляющие x, то для вычисления x в терме t строятся n подтермов t_1, ..., t_n , $top(t_i)=b_i$, i=1, ..., n. Таким образом, в построенном (V, W)-плане будет n термов, которые отличаются друг от друга подтермами, вычисляющими в них переменную x.

При реализации алгоритма термы следует строить не сверху — вниз, а, напротив, слева — направо, т. е. строить полностью одну (самую левую) ветвь дерева, представляющего терм t, и лишь после этого переходить к построению другой. В этом случае проще будет избавиться с использованием стратегии Д от

незапланированных дублирующих вычислений. Предложенный алгоритм, естественно, не гарантирует построения такого (*V*,*W*)-плана, что на некоторой реализующей его ПП достигает экстремума достаточно сложный функционал. Однако он дает большие возможности пользователю для управления процессом построения (*V*,*W*)-плана с помощью такого вида директив:

1)указать список операций, употребление которых в (*V*,*W*)-плане недопустимо;

2)указать переменные, которые не должны использоваться в (*V*,*W*)-плане;

3)указать список модулей, употребление которых в ПП нежелательно (по этой директиве операция, интерпретация которой задается одним из этих модулей, выбирается лишь в том случае, если не может быть выбрана другая);

4)указать переменные и операции, использование которых в (*V*,*W*)-плане желательно;

5)указать стратегию выбора операции;

6)запретить использование каких-либо процессоров мультикомпьютера, т. е. требуется не использовать в (*V*,*W*)-плане операций, которые могут быть выполнены только на запрещенных процессорах;

7)потребовать *n*-кратного дублирования вычисления некоторой переменной *x* для повышения надежности ПП. В (*V*,*W*)-плане по этой директиве появится *n* подтермов,

вычисляющих x (если удастся столько построить), а в синтезированной ПП значения, выработанные разными подтермами, сравниваются. Если эти значения окажутся не равными, то, следовательно, либо входные данные программы, либо модули содержат ошибку, о чем ПП должна выдать сообщение;

8) использовать для вычисления переменной x условную операцию a. При выборе операции, вычисляющей x, алгоритм всегда берет безусловную операцию, а если такой нет - то все условные операции. Такое решение плохо тем, что интерпретация безусловной операции задается универсальным модулем, который должен работать, вообще говоря, хуже, чем специализированный. Поэтому в (V,W)-плане полезно использовать для вычисления переменной x и условную, и безусловную операции, если из анализа входных данных следует, что значение x будет достаточно часто вычисляться специализированным модулем.

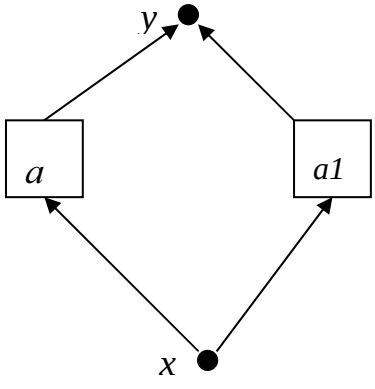


Рис. 7.20

На рис. 7.20 показана ПВМ. Пусть модуль mod_a, вычисляющий функцию $f_a=\boldsymbol{I}(a)$, вырабатывает логическое значение y=**true**, если значение x равно 7, в противном случае **false**, а универсальный модуль mod_{a1} вырабатывает значение y=**true** всякий раз, когда значением x является простое число, иначе **false**. Операция a описана как условная, т.е., **if** x=7 **then** $a(x)$; Множество $T^W{}_V$, $V=\{x\}$, $W=\{y\}$, содержит два терма $a(x)$ и $a_1(x)$, могут быть построены два ($\{x\},\{y\}$)-плана: $T_1=\{a1(x)\}$ и $T_2=\{a(x),a1(x)\}$. Тогда программа Π_{T2} вида

if x=7 **then** (y:=$a(x)$; **go to** L); y:=$a1(x)$; L: ...

будет работать в среднем быстрее, чем Π_{T1}, если она многократно выполняется и среди входных значений Π_{T2} достаточно часто встречается число 7. По этой же директиве можно требовать использования нескольких условных операций для вычисления x и называть порядок их выполнения в ПП.

Список подобных директив может быть продолжен. Кроме того, каждая директива может иметь уточняющие параметры, например, потребовать применять стратегию **Б** только для построения подтерма, вычисляющего переменную x или запретить использовать операцию a для вычисления только переменной y, или потребовать вычислять x только в терме t операцией b, и т. п. Заметим, что применение некоторых стратегий приводит к экспоненциальной сложности алгоритма

выбора, а это означает, что надо иметь возможность так локализовать применение этих стратегий (строить с их помощью не весь (*V*,*W*)-план, а только, например, некоторый подтерм), чтобы это не приводило к неприемлемо большому перебору. Уточняющие параметры как раз и позволяют реализовать эту возможность. Вся информация, необходимая для управления конструированием (*V*,*W*)-плана, может быть получена либо из знаний о ПО, либо из анализа входных данных и результатов тестовых испытаний синтезированной ПП при решении конкретных задач.

Для того чтобы можно было реализовать в ССПП группу директив, требующих обязательного использования в конструируемом (*V*,*W*)-плане *T* некоторых переменных и операций, необходимо несколько модифицировать алгоритмы планирования и выбора. Эти модификации рассмотрим на примере построения (*V*,*W*)-плана *T*, в котором требуется использовать переменную *x*.

Пусть в восходящей части алгоритма планирования обнаружилось, что $x \in V_i$ при некотором $i<k$ (обозначения те же, что введены при описании алгоритма планирования). Невыполнение этого условия означает, что требуемый (*V*,*W*)-план *T* просто не существует и ни в одном терме из T_V^W не используется переменная *x*. Далее, в восходящей части алгоритма планирования наряду с множеством F_i строится множество

$$F_i^x=\{a\in F_i \mid x\in in(a)\}=\{a\in A(x) \mid in(a)\subseteq V_i\},$$

затем V_{i+1} и $V_{i+1}^x=\{y\in \mathbf{X} \mid y\in out(a)\wedge a\in F_i^x\}$,

$F^x{}_{i+1}=\{a\in F_{i+1} \mid in(a)\cap V_{i+1}^x\neq\varnothing\}$ и т. д. Ясно теперь, что алгоритмом выбора требуемый (V,W)-план T будет построен, если при выборе операции предпочтение отдавать операциям из множества

$$F^x=\bigcup_{j\in\{i,\ldots,k\}} F_j^x$$

7.2.4.Структурированные операции и их преобразования

В предыдущем разделе рассмотрены основные алгоритмы конструирования (V,W)-плана T. С их помощью уже можно автоматически либо в интерактивном режиме строить и целенаправленно модифицировать план T, если реализующая его ПП будет признана неудовлетворительной по результатам тестовых испытаний. Заметим, что эти модификации будут носить “внешний” характер, позволяя запрещать либо требовать использовать в T те или иные операции. Изменить же “внутреннее содержание” операций указанными алгоритмами нельзя, а изменения такие необходимы для проведения более тонких оптимизирующих преобразований T.

Отметим прежде всего, что для конструирования качественных ПП в ССПП нужно уметь выбирать наиболее подходящий в каждом конкретном случае алгоритм вычисления функции $f_a=I(a)$, а не фиксировать его, используя для вычисления

f_a модуль mod_a. Такая фиксация уменьшает мобильность (переносимость) ПП, что совершенно недопустимо, так как окажется невозможным накопление программного фонда для мультикомпьютеров в виду большого разнообразия в их архитектуре. Действительно, сконструированная ПП, в которой есть обращение к mod_a, не сможет выполняться на мультикомпьютере, если в нем нет всех необходимых ресурсов. Для обеспечения мобильности ПП в этом случае надо уметь либо автоматически преобразовывать mod_a, что далеко не всегда можно сделать, либо использовать другой модуль вместо mod_a. В общем случае необходимо синтезировать подходящий алгоритм для вычисления f_a и реализующий его mod_a.

Нужно уметь также изменять интерпретацию некоторых операций. Пусть, например, на ПВМ C определен (V,W)-план T, задающий алгоритм вычисления определенного интеграла с подынтегральной функцией $f_a = I(a)$, операция a используется в T. Понятно, что нет смысла для каждой вновь требуемой подынтегральной функции создавать свою ПВМ, достаточно иметь возможность менять интерпретацию операции a. Именно так делается при обращении к стандартной программе вычисления интеграла, которой в качестве параметра передается имя процедуры, вычисляющей нужную в данном конкретном случае подынтегральную функцию.

Для решения указанных задач используется понятие

структурной интерпретации ПВМ. Пусть G — конечное множество ПВМ, в каждой ПВМ $B=(X_B,F_B)$, $B\in G$, существует (V_B,W_B)-план T_B. *Структурной интерпретацией* ПВМ $C=(X_C,F_C)$ называется функция SI, которая сопоставляет:

1)каждой операции b, $b\in F_C$, – вычислительную модель $B=SI(b)$, $B\in G$;

2)каждой переменной $x\in in(b)$ – переменную $v=SI(x)$, $v\in V_B$; каждой переменной $y\in out(b)$ – переменную $w=SI(y)$, $w\in W_B$, и, кроме того,

$$\forall u\,(u\in((V_B\cup(W_B)\rightarrow\exists x(x\in in(a)\cup out(a)\wedge u=SI(x))).$$

Операция b называется *структурированной операцией* и говорится, что операция b *раскрывается* ПВМ B. Далее предполагается также, что в B нет операции b и при раскрытии структурированных операций B никогда не встретится операция b. Таким образом, все ПВМ из G должны быть описаны до момента определения структурированной операции b. Здесь это чисто техническое ограничение, введенное для упрощения описаний[1]. В отличие от структурной интерпретации SI интерпретация I будет иногда называться *содержательной интерпретацией* для однозначности формулировок. Понятие интерпретации (содержательной) ПВМ доопределяется

[1] Если снять это ограничение, то будет определен класс рекурсивных ВМ [5]. В контексте текущего рассмотрения они мало интересны и потому не обсуждаются.

естественным образом.

Пусть заданы интерпретация I_B ПВМ B в области интерпретации D_B и интерпретация I_C ПВМ C в области интерпретации D_C, причем функция $f_b = I_C(b)$ вычисляется алгоритмом, заданным (V_B,W_B)-планом T_B. Если $t=b(t_i,...,t_n)$ — некоторый терм из множества T_V^W, построенного на ПВМ C, $in(b)=\{x_1,...,x_n\}$, $out(b)=\{y_1,...,y_m\}$, то при вычислении $d_{y_i}=val(t,y_i)$, $i=1, ..., m$, сначала вычисляются $d_{x_j}=val(t_j,x_j)$, $j=1, ..., n$, переменные $v_j=SI(x_j)$ получают значения d_{x_j}, а затем $d_{w_i}=val(t_i',w_i)$, $t_i' \in T_B$, $w_i=SI(y_i)$, и переменные y_i получают значения d_{w_i}, которые и обозначаются d_{y_i}.

Для корректности этого доопределения необходимо, чтобы выполнялось условие $\{d_{x_1}, ..., d_{x_n}, d_{w_1}, ..., d_{w_m}\} \subseteq D_C \cap D_B$. Интерпретации I_C и I_B называются *согласованными.* Таким образом, структурная интерпретация $SI(b)=B$ задает внутреннюю структуру операции b, а в качестве алгоритма вычисления функции $f_b=I_C(b)$ может быть взят любой (V_B,W_B)-план, построенный па ПВМ B.

Раскрытием b называется построение множества T_B. При раскрытии b могут выполняться оптимизирующие преобразования структурированной операции. Рассмотрим

несколько таких преобразований. Введенные в начале параграфа обозначения сохраняют свой смысл. Предполагается также, что ПВМ $C=(X_C,F_C)$ и $B=(X_B,F_B)$ представлены таблицами TX_C и $ОП_C$, TX_B и $ОП_B$ соответственно и, кроме того, операция b используется в некотором (V,W)-плане T, построенном на C. При всех преобразованиях структурная интерпретация переменных из X_C не изменяется.

Открытая подстановка. Строится ПВМ $C1=(X1,F1)$, $ОП_{C1}=ОП_C \oplus ОП_B$, $TX_{C1}=TX_C \oplus TX_B$, значок $\oplus$ обозначает объединение таблиц, при этом если $u=SI(z)$, и $u \in V_B \cup W_B$, $z \in (in(b) \cup out(b))$, то переменная u переименовывается и получает имя z везде, где u встречается в TX_{C1} и $ОП_{C1}$. Строка u из TX_{C1} вычеркивается, а в строке z множество $A(z)=\{a \in (\mathbf{F}_C \cup \mathbf{F}_B) \mid z \in in(a)\}$, а $comp(z)=\{a \in (F_C \cup F_B) \mid z \in out(a)\}$. Строка b также вычеркивается из $ОП_{C1}$, а операция b удаляется из всех множеств. Предполагается, что в С и в В нет одноименных переменных и операций. Такое раскрытие аналогично открытой подстановке макроопределения вместо макровызова. При планировании на С1 будет построено и множество T_B.

Раскрытие структурированной операции b открытой подстановкой возможно не всегда, в частности, тогда, когда функция f_a должна в синтезированной ПП вычисляться процедурой. В этом случае планирование на ПВМ В (построение T_B) ведется отдельно от планирования на ПВМ С.

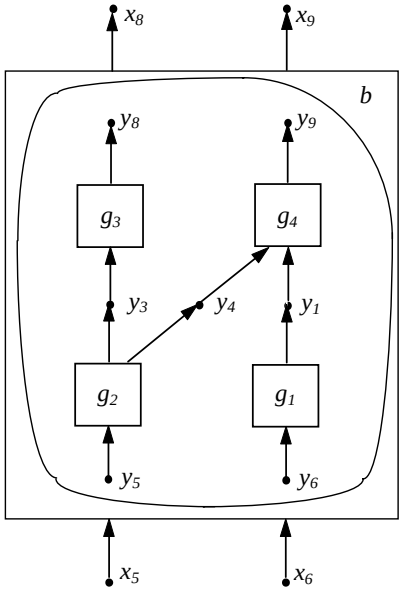


Рис. 7.21.

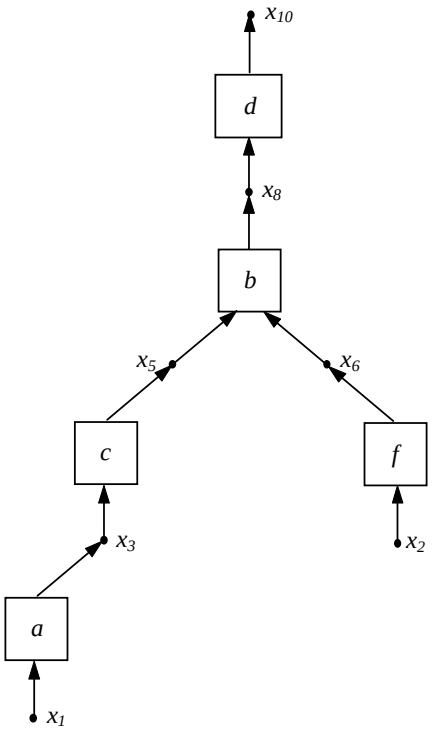


Рис. 7.22.

Чистка структурированной операции. Пусть в T используется лишь часть $\underline{out}(b)$ выходных переменных операций b, т. е. выполняется строгое включение $\underline{out}(b) \subset out(b)$. Множество переменных ПВМ B, которые соответствуют в структурной интерпретации переменным из $\underline{\text{out}}(b)$, обозначим $\underline{W_B}$. Тогда из TX_B и ОП_B могут быть в нисходящей части алгоритма планирования на B удалены переменные и операции, которые не используются в термах, вычисляющих $\underline{W_B}$, при этом вместо V_B

может быть взято меньшее в общем случае множество $\underline{V_B} \subseteq V_B$. Такая чистка b делается описанным алгоритмом планирования и может существенно улучшить ПП, реализующую T, так как из термов T можно удалить подтермы, вычисляющие переменные из множества $in(b) \backslash \underline{in}(b)$ (через $\underline{in}(b)$ обозначено множество тех переменных из $\mathrm{in}(b)$, которым в структурной интерпретации соответствуют переменные из $\underline{V_B}$). Кроме того, в ПП вместо процедуры, реализующей (V_B, W_B)-план T_B, может быть использована процедура, реализующая $(\underline{V_B}, \underline{W_B})$-план T_B для которого справедливо строгое включение $\underline{T_B} \subset T_B$.

На рис. 7.21 приведена ПВМ B, раскрывающая структурированную операцию b ПВМ, показанной на рис. 7.17, терм t на рис. 7.22 вычисляет переменную x_{10} из переменных $\{x_1, x_2\}$. Так как в t для вычисления x_{10} используется из двух выходных переменных операции b только одна переменная x_8, то при раскрытии b строятся только термы, вычисляющие $y_8 = \mathrm{SI}(x_8)$, в данном случае построится терм $y_5 \to g_2 \to y_3 \to g_3 \to y_8$ и, следовательно, переменная y_6 не нужна для вычисления y_8. После чистки операции b образуется операция b', $\mathrm{in}(b') = \{x_5\}$, $\mathrm{out}(b') = \{x_8\}$, и в t становится ненужным подтерм $x_2 \to f \to x_6$. В результате переменная x_{10} будет вычисляться более простым, чем t, термом $x_1 \to a \to x_3 \to c \to x_5 \to b' \to x_8 \to d \to x_{10}$.

Выделение подопераций. В ПП, реализующей (V,W)-план T, нет в общем случае необходимости ожидать вычисления значений

всех переменных из in(b) для того, чтобы начать выполнение mod_b, реализующего (V_B,W_B)-план T_B. Выполнение mod_b может начаться, когда известны значения лишь части переменных из in(b) (*смешанные* или *частичные* вычисления). Это можно делать динамически, но здесь рассматриваются только статические методы конструирования программ и потому частичность вычислений выливается в изменений структуры ВМ.

Для конструирования такого mod_b используется прием выделения подопераций, когда b разделяется на несколько структурированных операций. Выделение подопераций производится в процессе планирования на ПЭВМ C.

Алгоритм планирования модифицируется следующим образом. Всякий раз, когда завершено построение множества V_i, i=0, ..., k, в восходящей части алгоритма планирования строится множество $V_{b_i}=V_i \cap in(b)$. Если оно пусто, то планирование на C продолжается дальше, если же нет — планирование на C прерывается, образуется множество $V_{b_{1,0}} = \bigcup_{x \in V_{b_1}} SI(x)$, и начинается планирование на B (восходящая часть алгоритма), при котором строятся множества переменных $V b_i, j_1$ и операций $F b_i, j_i$, j_1=0, ..., k_1; k_1 таково, что $F b_1, k_1 = \varnothing$. В этот момент планирование на B завершается, образуется структурированная операция b_1, $SI(b_1)=B1$, где $B1$ — есть ПВМ, определяемая всеми

переменными и операциями из множеств $Vb_{i,}\ j_1$ и $Fb_{i,}\ j_1$, т. е.

$$B1=\left(\left(\bigcup_{j_1} V_{b_1,j_1}\right),\ \left(\bigcup_{j_1} F_{b_1,j_1}\right)\right).$$

Множество in(b_1) содержит все переменные из Vb_1, а out(b_1) — все переменные $y\in X_C$ такие, что $SI(y)\in Wb_1$, где

$$W_{b_1}=\left(\left(\bigcup_{j_1=1}^{k_1} V_{b_1,j_1}\right)\setminus V_{b_1,0}\right)$$

и все переменные из Wb_1. Из F_b удаляются все операции, попавшие в множество $\bigcup_{j_1=0}^{k_1} F_{b_1,j_1}$. Структурированная операция b_1 называется подоперацией структурированной операции b. Операция b_1 добавляется в $\boldsymbol{F}_C$ и F_l, переменные out(b_1) — в X_C и в V_{i+1} (при необходимости переменные из Wb_1 могут быть переименованы).

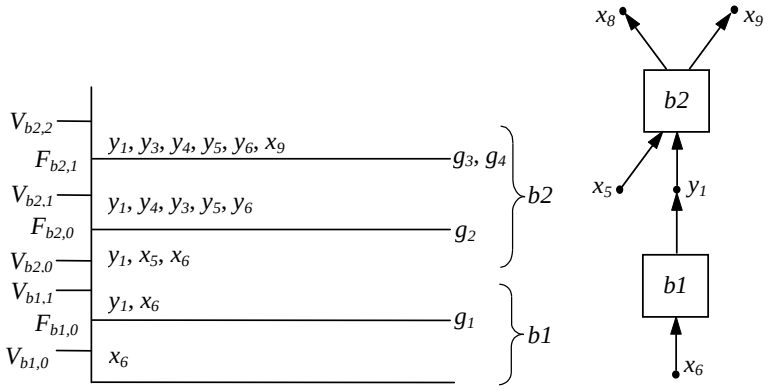
Если $z\in W_b$, то считается, что $SI(z)=z$. После этого возобновляется планирование на C с прерванного места. Если при некотором l, $k>l>i$, окажется, что множество $Vb_2=(V_l\cap in(b))\setminus(in(b_1)\cup out(b_1))$ непустое, то вновь следует прервать планирование на C; образовать множество переменных

$$V_{b_2,0}=\left(\bigcup_{x\in V_{b_2}} SI\ (x)\right)\cup\left(W_{b_1}\cup V_{b_1,0}\right);$$

построить множества V_{b_2,j_2} и F_{b_2,j_2}, j=0, ..., k_2, $F_{b_2,k_2} = \varnothing$; образовать подоперацию b_2, $SI(b_2)=B2$, где ПВМ $B2$ определена всеми переменными и операциями из множеств V_{b_2,j_2} и F_{b_2,j_2}, in(b_2) содержат все переменные из $V_{b_2} \cup out(b_1) \cup in(b_1)$, а $out(b_2)$ — все переменные $y \in \mathbf{X_C}$ такие, что $(SI(y) \in W_{b_2}) \wedge (y \notin (in(b_1) \cup out(b_1)))$, и все переменные W_{b_2},

$$W_{b_2} = \left(\left(\bigcup_{j_2=1}^{k_2} V_{b_2,j_2} \right) \setminus \left(W_{b_1} \cup V_{b_1,0} \cup V_{b_2,0} \right) \right) .$$

Операция b_2 добавляется в F_C и F_l переменные $out(b_2)$ - в X_C и V_{l+1}. Если $y \in W_{b_2}$, то $SI(y)=y$, $I(b_2)$ задается $(V_{b_2,0}, W_{b_2})$-планом, из F_B удаляются операции, вошедшие в множество $\bigcup_{j_2=0}^{k_2} F_{b_2,j_2}$. Затем возобновляется планирование на C и т. д. После завершения восходящей части алгоритма планирования на C из TX_C и $ОП_C$ удаляется b. В результате после планирования в таблицах TX_C и $ОП_C$ появляются подоперации структурированной операции b, если, конечно, они не будут удалены в нисходящей части алгоритма планирования на C. Для всех оставшихся в таблицах структурированных операций после завершения планирования (когда станет известно, какие из их выходных переменных используются в термах из T_V^W) проводится чистка.



а)

рис. 7.23

Пример выделения подопераций показан на рис. 7.23. В результате выделения подопераций (рис. 7.23а) из ПВМ на рис. 7.17 будет удалена операция b, а вместо нее появятся операции b_1, b_2 (рис. 7.23б), что позволит построить ПП, в которой вычисления “внутри” b можно начинать сразу же, как только это станет возможным. Поскольку для вычисления x_{10} (см. рис. 7.17) нужна только переменная $x \in \text{out}(b)$, то в нисходящей части алгоритма планирования проводится чистка b_2, в результате которой из операции b_2 (имеется в виду из F_{b_2}, где $SI(b_2)=B2=(X_{b_2},F_{b_2})$) удаляется операция g_4, вычисляющая y_9, а для оставшейся после чистки операции b_2', $\text{in}(b_2')=\{x_5\}$, $\text{out}(b_2')=\{x_8\}$, не требуется вычислять переменную y_1, и операция b_1 оказывается не нужной для вычисления x_{10} (не используется в

терме, вычисляющем x_{10}, рис. 7.24).

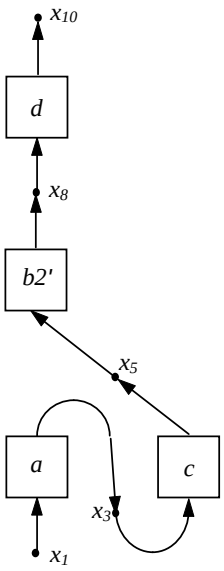


рис. 7.24

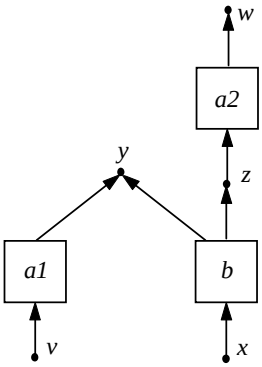


рис. 7.25

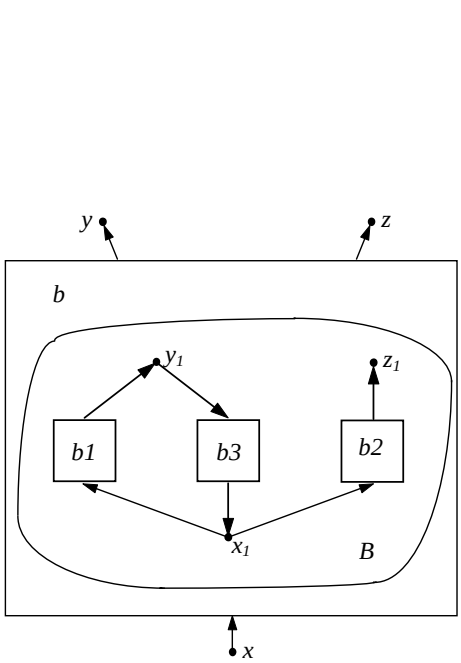


Рис. 7.26

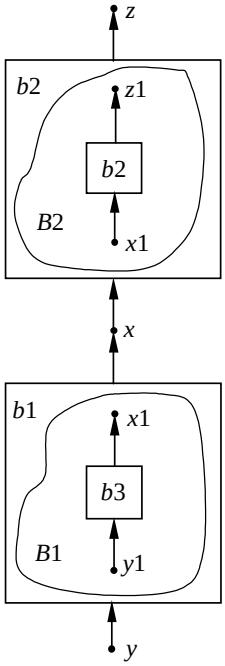


Рис. 7

Наряду с $(V_{b,0},W_b)$ -планом в B может, к примеру, существовать и $(W_b,V_{b,0})$-план, и в этом случае было бы возможно вычислять переменные in(b) из переменных out(b). Поэтому основанием для образования подоперации b_1 лучше считать непустоту множества $V'_{b_1}=V_i\cap(in(b)\cup out(b))$ (этот критерий называется *критерием планировщика*), и его же переменные должны войти в $in(b_1)$. Аналогичным образом определяются и множества V'_{b_m}, m = 2, 3, ... Такое решение позволяет строить новые алгоритмы

вычисления W из V, его полезность иллюстрирует следующий пример.

Пусть в ПВМ C (рис. 7.25) есть структурированная операция b, $SI(b)=B$ (ПВМ B показана на рис. 7.26), $in(b)=\{x\}$, $out(b)=\{y, z\}$, $SI(x)=\{x_1\}$, $SI(y)=\{y_1\}$, $SI(z)=\{z_1\}$.

Если на C поставлена задача построения $(\{v\},\{w\})$-плана, то она может быть решена только выделением подопераций b'_1 и b_2', $in(b_1')=\{y\}$, $out(b'_2)=\{z\}$, $out(b_1')=in(b'_2)=\{x\}$ (рис. 7.27).

Для выделения подопераций кроме критерия планировщика полезно использовать следующие критерии.

А. Раннее время вычисления значений переменных. В соответствии с этим критерием в V_{b_1} попадают те переменные из in(b), значения которых вычислились раньше при тестовом испытании синтезированной ПП, использующей mod_b. Алгоритм выделения подопераций по данному критерию строится очевидным образом.

Б. Нужные переменные. Пусть при тестовых испытаниях выяснилось, что задержки в выполнении модулей ПП и простой процессоров обусловлены долгим вычислением значения переменной x. Если она вычисляется структурированной операцией b, то можно выделить подоперацию b_1, которая вычисляет только переменную x, и при компиляции ПП назначить ее на выполнение раньше, чем оставшуюся

подоперацию b_2. Для этого в T_{b_1} отбираются все термы, вычисляющие $SI\{x\}$, и образуется подоперация b_1; $in(b_1)$ содержит все переменные из множества

$$\{y|SI(y)\in in(t)\wedge(t\in T_{b_1})\wedge(SI(x)\in out(t))\};$$

ПВМ $B1=(X_{B1},F_{B1})$ Определена всеми переменными и операциями, используемыми в T_{B1}.

ПВМ $B1$ строится следующим образом. Сначала в X_{B1} попадает переменная $SI(x)$, а в F_{B1} — все операции из множества comp ($SI(x)$). Затем в X_{B1} добавляется множество $In(comp(SI(x)))$, объединяющее все входные переменные операций из comp($SI(x)$), а в F_{B1} - все новые операции из множества $\bigcup_{y\in X_{B1}} \text{comp}(y)$, т.е. все операции из $\left(\bigcup_{y\in X_{B1}} \text{comp}(y)\right)\backslash \mathbf{F_{B1}}$, и так до тех пор, пока на некотором шаге новых операций не окажется.

Аналогичным образом строится ПВМ $B2$, $SI(b_2)=B2$, $out(b_2)=out(b)\backslash\{x\}$. Если существуют термы $t_1\in T_{\mathbf{B1}}$ и $t_2\in T_{\mathbf{B2}}$ такие, что $sbt(t_1)\cap sbt(t_2)\neq\varnothing$, то может быть выделена подоперация b_{12} подопераций b_1 и b_2, причем алгоритм вычисления функции $I(b_{12})$ задается общими для t_1 и t_2 подтермами. Общие подтермы выявляются при построении $T_{\mathbf{B1}}$ и $T_{\mathbf{B2}}$ алгоритмом выбора (V,W)-плана;

В. Чистка циклов. Пусть в циклическом участке синтезированной ПП есть обращение к mod_b, вычисляющем

функцию $f_a=I(b)$, и b — структурированная операция, $SI(b)=B$**.** Если в множестве in(b) есть переменные y_1, ..., y_m, значения которых вычисляются за пределами циклического участка и не перевычисляются внутри него, то из b могут быть удалены не массовые вычисления. Для этого следующим образом выделяется подоперация b_1:

-формируется множество переменных $V_0=\{z|z=SI(y_i),\ i=1,\ ...,\ m\}$;

-проводится восходящая часть алгоритма планирования на B и строятся множества переменных $V^* = \bigcup_{i=0}^{k} V_i$ и операций $F^*= \bigcup_{i=0}^{k} F_i$, где $F_k=\varnothing$;

-образуется подоперация b_1, $SI(b_1)=B1$**,** где $B1=(V^*,F^*)$, $in(b_1)=\{y_1, ..., y_m)$, $out(b_1)=V^*\backslash in(b_1)$;

-из множества F_b удаляются операции F^*, после этого производится чистка структурированной операции b.

Таким образом, в b_1 попадают из b только не массовые вычисления, которые повторялись в каждой итерации цикла. Для управления выделением подопераций можно предложить следующие директивы:

- задать критерий выделения подопераций; эта директива может иметь различные параметры, например, может

указываться, что она применима ко всем структурированным операциям, либо только к конкретной, либо только к тем, которые назначены на некоторый процессор;

- указать количество подопераций, которые необходимо выделить, например, выделить по критерию планировщика не более трех подопераций структурированной операции b;
- указать список переменных из out(b), для вычисления которых необходимо выделить подоперацию структурированной операции.

Объединение структурированных операций. Это преобразование применяется для того, чтобы уменьшить затраты ресурсов на реализацию управления в ПП и сделать операции, используемые в T, более “однородными”. Пусть b_1 и b_2 - структурированные операции, $SI(b_1)=B1$, $SI(b_2)=B2$. При объединении b_1 и b_2 образуется структурированная операция b, $in(b)=in(b_1)\cup in(b_2)$, $out(b)=(out(b_1)\cup out(b_2))\setminus(in(b_1)\cup in(b_2))$, $SI(b)=B$. Здесь ПВМ B определена всеми переменными и операциями $B1$ и $B2$, причем если переменная $x\in(in(b_1)\cup out(b_1))\cap(in(b_2)\cup out(b_2))$, то все переменные $B1$ и $B2$, соответствующие x в структурной интерпретации, отождествляются в B друг с другом (имеют одно и то же имя). На основе описанных преобразований структурированных операций в ССПП можно очевидным образом реализовать ряд директив

высокого уровня, позволяющих человеку оптимизировать алгоритм решения сформулированной задачи.

В языке для задания структурной интерпретации можно использовать конструкцию оператора постановки задачи. Например, описание

b=(**на** B **вычислить** $w_1,...,w_m$ **из** $v_1, ..., v_n$) **вх** $x_1, ..., x_n$ **вых** $y_1, ..., y_m$;

определяет структурированную операцию b, $SI(b)=B$, $SI(x_i)=v_i$, $SI(y_j)=w_j$, $i=1, ..., n$, $j=1, ..., m$. Изменение интерпретации некоторой операции b ПВМ C, необходимое для решения сформулированной задачи, можно сделать при формулировке задачи, например, так:

на С (b=(**на** D **вычислить...**)) **вычислить** w из v;

В этом описании определяется (а может быть, и переопределяется) структурная интерпретация операции b, $b \in F_C$. Заметим в заключение, что при определении структурированной операции нет необходимости выделять входные и выходные переменные. При планировании, в процессе выделения подопераций по критерию планировщика, входными переменными объявляются переменные из V_b, а выходными - переменные $\{y \in X_C | SI(y) \in W_b\}$. При решении различных задач различные переменные будут объявляться входными (выходными). В определении структурированной операции поэтому можно указывать просто списки переменных, например:

b=(**на** C **вычислить переменные** v_1, ..., v_{m+n}) **вхвых** x_1, ..., x_{m+n};.

7.2.5.Проблема компиляции параллельной программы

Конструирование ПП на вычислительных моделях состоит из построения (V,W)-плана T (выбор алгоритма) и записи на некотором алгоритмическом языке ПП $П_T$, реализующей T (компиляция ПП).

Выбор T и компиляцию $П_T$ необходимо сделать таким образом, чтобы функционал Ф, который определяет качество ПП и задан на множестве всех ПП, реализующих всевозможные (V,W)-планы, достигал экстремума на $П_T$. Лишь для некоторых простых функционалов Ф (например, глубина термов (V,W)-плана) задача конструирования оптимальной ПП может быть решена за полиномиальное время. Для большинства практически важных функционалов, таких, например, как время выполнения ПП, решение этой задачи имеет экспоненциальную сложность, оно зависит в общем случае от свойств мультикомпьютера, на которой должна выполняться сконструированная ПП, от свойств функционала Ф, от свойств выбранного алгоритма решения сформулированной задачи.

Кроме того, для оптимального решения сформулированной на ПВМ задачи при различных входных данных (значениях входных переменных) в общем случае необходимо использовать

различные алгоритмы T и реализующие их ПП (имеется в виду, что для разных входных данных функционал Ф достигает экстремума на различных ПП из множества $\bigcup_{T \in TVW}$ ПT). Информация о свойствах интерпретации в ПВМ отсутствует вообще, поэтому при конструировании ПП необходима помощь человека для построения высококачественных ПП. Таким образом, на последних двух этапах в ССПП следует использовать человеко-машинные алгоритмы конструирования ПП, причем необходимая для решения сформулированной на ПВМ задачи ПП должна конструироваться в ССПП и без участия пользователя. Однако в случае, когда качество автоматически синтезированной ПП не удовлетворяет пользователя, ему необходимо будет дать ССПП дополнительную информацию и затратить дополнительные усилия для конструирования ПП более высокого качества. Качество сконструированной ПП оценивается пользователем по результатам ее испытания на серии тестов (тестовое испытание ПП).

Таким образом, автоматически синтезировать на ПВМ оптимальную относительно какого-либо практически важного и достаточно сложного функционала ПП в общем случае не удается. А раз так, то для построения ПП следует использовать прагматический подход, который состоит в том, чтобы сначала автоматически конструировать оптимальную ПП, но относительно достаточно простого функционала. Если эта ПП не

удовлетворяет пользователя, он должен иметь возможности для проведения необходимых оптимизирующих преобразований T и ПП $П_T$. Для этого в ССПП должны быть директивы различного уровня вплоть до прямых указаний, например, о том, как должны быть распределены ресурсы мультикомпьютера и какова структура ПП. Такой подход к синтезу ПП может быть в ССПП реализован следующим образом.

На первом шаге конструирования ПП выбирается некоторый (V,W)-план T, например произвольный или построенный по какой-нибудь другой стратегии. Если при создании описания ПО проследить за тем, чтобы в ПВМ не содержались заведомо неудачные алгоритмы (это должно быть одной из задач отладки описания ПО), то даже произвольный T нередко будет приемлемым. При разработке пакетов прикладных программ именно такие описания ПО и создаются. Далее необходимо распределить ресурсы мультикомпьютера и, прежде всего, назначить операции на процессоры мультикомпьютера. После выполнения алгоритма назначения каждому процессору мультикомпьютера сопоставляется (статически или динамически) множество операций, которые должны быть реализованы на нем.

На следующем шаге производится “запись” ПП на некотором языке. Эта ПП состоит из обращений к модулям в любом допустимом порядке, память распределяется по известным в теории компиляции алгоритмам. Для последовательных

фрагментов ПП могут быть проведены оптимизирующие преобразования. Конечно, для разных классов мультикомпьютеров в ССПП должны использоваться разные алгоритмы конструирования произвольной ПП и это означает, что мобильность синтезированной ПП обеспечивается переносом ССПП на новый мультикомпьютер. Представление алгоритмов в виде множества термов *T* не зависит от свойств мультикомпьютера и именно в такой машинно-независимой форме происходит накопление фонда алгоритмов для мультикомпьютеров.

Автоматически сконструированная ПП называется произвольной, это и есть оптимальная ПП относительно какого-то функционала, точно определять и вычислять который нет необходимости. Высокое качество произвольной ПП, естественно, гарантировать нельзя, и ее придется модифицировать, если свойства ПП (время выполнения, требуемые ресурсы, надежность и т. п.) не удовлетворяют пользователя, сформулировавшего задачу. Модификации подвергаются как реализуемый ПП алгоритм *T*, так и структура ПП (назначение модулей на те или иные процессоры мультикомпьютера для выполнения, распределение памяти и других ресурсов, управление). Очень важно, что в ССПП такие преобразования могут делаться зачастую средствами высокого уровня и в случае, когда требования к качеству ПП не слишком

высоки, их использование не потребует от пользователя существенно бо̀льших знаний, чем это было нужно для формулировки его задачи. Конечно, при повышении требований к качеству ПП от пользователя потребуются и большие знания как о ПО, так и о мультикомпьютере.

Те или иные модификации выбираются на основе анализа результатов тестовых испытаний ПП, необходимая информация собирается системными модулями, включаемыми в ПП при ее конструировании. Некоторые преобразования имеет смысл делать уже при конструировании произвольной ПП, в первую очередь, чистку структурированных операций, выделение подопераций по критерию планировщика, набор стратегий для набора (*V*,*W*)-плана.

Другая часть преобразований может осуществляться автоматически, т. е. решение об их выполнении принимает ССПП, но под контролем человека. Например, из результатов тестовых испытаний ПП выяснилась возможность лучше распределить ресурсы мультикомпьютера, но только пользователь может решить, повторится ли такая •ситуация на тех входных данных, с которыми будет работать ПП. Третья часть преобразований может быть выполнена только по прямым директивам пользователя, так как только он обладает информацией, необходимой для принятия решения о преобразовании *T* и ПП. К их числу относятся требования о

включении или не включении условных операций в *T*, об обеспечении требуемой степени надежности вычисления каких-либо переменных и ПП в целом.

7.3. ВЫЧИСЛИТЕЛЬНЫЕ МОДЕЛИ С МАССИВАМИ

Метод синтеза ПП на ПВМ часто не позволяет конструировать эффективные ПП в силу того, что в них нельзя задавать массовое применение операций. Это приводит к тому, что в ПП все массовые вычисления должны выполняться внутри модулей. А так как основное время решения задач приходится именно на них, то для уменьшения времени исполнения ПП такие модули надо распараллеливать, что далеко не всегда можно сделать автоматически. По этой же причине модуль, предназначенный для массового применения (к примеру, модуль считывания одной записи файла), не может быть непосредственно включен в ПВМ. Его обязательно надо включать в объемлющий модуль, в котором и должно быть задано управление, определяющее массовое вычисление. Эти и некоторые другие обстоятельства обусловили необходимость введения массивов в вычислительные модели.

Вычислительные модели с массивами (ВММ) определяются в общем случае довольно сложно. По этой причине вначале будут рассмотрены упрощенные ВММ (УВММ), где отсутствуют такие объекты, как счетчики и предикатные символы.

7.3.1.Упрощенные вычислительные модели с массивами

Пусть заданы следующие множества и функции.

1. Множество переменных $\mathbf{X}=X\cup Y$, где $X=\{x, y, ..., z\}$ — конечное множество, элементы X называются *простыми*

переменными.

Множество Y — счетное либо пустое; непустое множество Y разбито на конечное число счетных, непересекающихся, линейно упорядоченных подмножеств, которые называются *массивами.* Элементы массива $\overline{x}=\{x_1,...,x_n,...\}$ называются *компонентами* $\overline{x}$, компонент $\overline{x}_n$ обозначается $\overline{x}[n]$; множество всех массивов обозначается $\overline{Y}=\{\overline{x}, \overline{y}, ..., \overline{z}\}$, $X\cap Y=\varnothing$.

2. Множество $F\subseteq F_1\times N$, где N — множество натуральных чисел; $F_1=\{a, b, ..., c\}$ — конечное множество функциональных символов (*операций*). Элемент $(a,i)\in F$ обозначается a^i и называется i-м *экземпляром* a. Операция a называется *простой*, если $N_a=\{n\in N|(a,n)\in F\}$ — одноэлементное множество, и *массовой*, если N_a содержит более одного элемента. Функция In сопоставляет каждому экземпляру a^i единственное конечное множество входных $\mathrm{in}(a^i)=In(a,i)$, а функция Out — единственное конечное множество выходных $out(a^i)=Out(a,i)$ переменных,

$$in(a^i)\cup out(a^i)\subseteq X,\ in(a^i)\neq\varnothing,\ out(a^i)\neq\varnothing.$$

Если некоторое свойство выполняется для всех экземпляров операции a, то, скажем, что это свойство выполняется для операции a и наоборот. Будем также говорить, что операция применяется к переменным $\mathrm{in}(a^i)$ для *вычисления* переменных $out(a^i)$. Индекс i в обозначениях далее будет опускаться в случаях, когда номер экземпляра несуществен.

На множества $in(a^i)$ и $out(a^i)$ накладываются обычные для

программных модулей ограничения, а именно:

1) $in(a^i) \neq in(a^j)$;
2) $|in(a^i)| = |in(a^j)| \wedge |out(a^i)| = |out(a^j)|$;
3) множества $in(a^i)$ и $in(a^j)$, а также $out(a^i)$ и $out(a^j)$ могут различаться только компонентами одноименных массивов, т. е. не допускается, чтобы, например, $in(a^1)=\{\overline{x}[1], \overline{x}[3]\}$, а $in(a^2)=\{\overline{x}[2], \overline{y}[3]\}$.

Здесь $i \in N$, $j \in N$, $i \neq j$; запись $|in(a^i)|$ обозначает число элементов в множестве $in(a^i)$. Введенные определения формализуют обычный в модульном программировании прием, когда отлаженный программный модуль включается в новую программу и применяется либо к некоторому набору входных переменных для вычисления набора выходных, либо в цикле к каждому набору из множества наборов входных переменных для вычисления множества наборов выходных переменных.

Набор $C=(X,F,In,Out)$ называется *упрощенной вычислительной моделью с массивами*. На C обычным образом определяются понятия терма, множества термов T_V^W, которое в общем случае будет теперь потенциально бесконечным множеством, (V,W)-плана вычислений, лишь при определении T_V^W вместо операции следует использовать понятие экземпляра операции. Множества V и W могут быть и бесконечными, например V может содержать все компоненты массива $\overline{x}$ с четными номерами, т. е.

$\{\overline{x}[2], \overline{x}[4], ...\} \subseteq V, V \subseteq X, W \subseteq X, V \cap W = \varnothing$.

При интерпретации I УВММ C в области D каждой переменной $x \in X$ сопоставляется единственный элемент $d_x \in D$ (значение переменной x), а каждой операции $a \in F_1$ — функция $f_a = I(a)$, отображающая соответствующие входные множества в выходные. Рассматриваются лишь те интерпретации, которые удовлетворяют условию $I(out(a)) = f_a(I(in(a)))$. Здесь $I(in(a))$ обозначает множество $\{d_{x_1}, d_{x_2}, \ldots, d_{x_m}\}$ значений, сопоставленных в интерпретации I переменным $in(a) = \{x_1, ..., x_m\}$. Таков же смысл обозначения $I(out(a))$. Таким образом, переменным $out(a) = \{y_1, y_2, ..., y_m\}$ присваивается значение $\{d_{y_1}, d_{y_2}, \ldots, d_{y_m}\}$ терма $t = a(t_1, ..., t_n)$, а множество термов T_V^W определяет множество всех определенных на C алгоритмов вычисления значений переменных W из значений переменных V. На рис. 7.28*а* приведен пример УВММ, эта же УВММ показана на рис. 7.28*б*, на котором массивы для наглядности представлены целостными объектами.

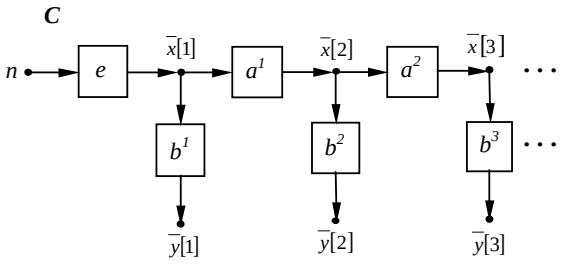


Рис. 7.28а.

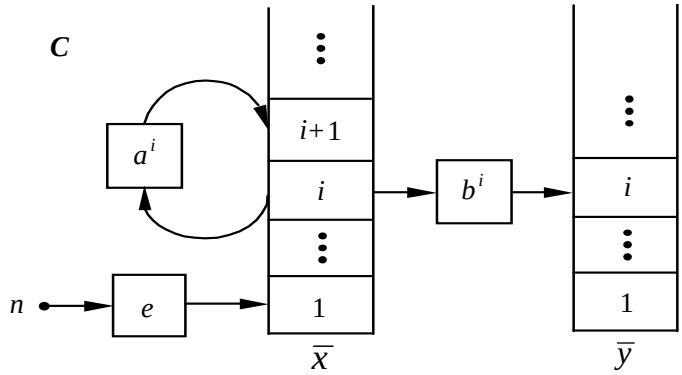


Рис. 7.28б.

Возможна следующая интерпретация C: функция $I(e)$ — считывание первой записи файла $\overline{x}$ в память ЭВМ; $I(a^i)$ — считывание i+1-й записи файла $\overline{x}$, $I(b^i)$ — вычисление i-й записи файла $\overline{y}$ из i-й записи файла $\overline{x}$.

Если $V=\{n\}$, $W=\{\overline{y}\}$, то (V, W)-план содержит бесконечное множество термов t_i вида $\overline{y}[i]=b^i(a^{i-1}(a^{i-2}(\ldots a^1(e(n)))$.

7.3.2. Динамические вычислительные модели с массивами

Понятно, что для ограничения вычислений нужны дополнительные средства, которые и вводятся в вычислительных моделях с массивами. Расширение УВММ делается в два шага. Вначале определяются динамические вычислительные модели с массивами (ВММ), а затем итеративные.

Пусть заданы:

1. Множество переменных $\mathbf{X}=X\cup Y\cup Z$, где X и Y — те же множества простых переменных и компонентов массивов, что и в УВММ, а Z — конечное множество переменных, называемых *счетчиками*, $Z\cap X=\varnothing$, $Z\cap Y=\varnothing$; каждому массиву $\overline{x} \in \overline{Y}$ соответствует счетчик из Z, который обозначается $n_{\overline{x}}$.

2. Множество $F\subseteq F_1\times N$ экземпляров операций, оно определяется так же, как и в УВММ, с тем дополнением, что входные и выходные наборы экземпляров операций могут содержать счетчики. Кроме того:

а) если a — массовая операция, то $\forall n_{\overline{x}} \in Z(n_{\overline{x}} \notin out(a) \& n_{\overline{x}} \in in(a).)$;

б) в множестве F существует только одна такая простая операции b, что $n_{\overline{x}} \in out(b)$.

Счетчики являются особыми переменными, значение счетчика $n_{\overline{x}}$ определяет число компонентов массива $\overline{x}$ (допускаются только такие интерпретации). Массивы $\overline{x} \in \overline{Y}$ называются *динамическими* массивами

Набор $M=(X,F,In,Out)$ называется *динамической вычислительной моделью с массивами* [5]. Пусть $V\subseteq X\cup Z\cup Y$, $W\subseteq X\cup Z\cup Y$, $V\cap W=\varnothing$. Предполагается, что если какие-либо компоненты массива $\overline{x}$ входят в V, то и $n_{\overline{x}} \in V$, это же справедливо и для W.

Понятие терма определяется обычным образом. В реализации

терма t необходимо будет учитывать, что в нем не может использоваться такой компонент $\overline{x}[i]$, что i превосходит значение, полученное счетчиком $n_{\overline{x}}$.

Понятно (см. главу 1), что в динамических моделях может быть задан алгоритм вычисления любой примитивно рекурсивной функции. Число компонентов массива заранее не определено, но в ходе вычислений значение счетчика определяется до начала использования компонентов массива.

7.3.3. Итеративные вычислительные модели с массивами

Понятно, что для представления в ВММ алгоритма вычисления любой частично рекурсивной функции в неё необходимо, кроме динамических, добавить ещё и *итеративные* массивы, которые бы допускали введение предикатов-предусловий операций и возможность задания счетчикам различных значений, чтобы число компонентов массива могло расти в ходе вычислений (как это нужно для определения оператора минимизации). Такое расширение делается довольно сложно технически, а поэтому итеративные ВММ демонстрируются на примерах. В примерах предполается, что понятия структурной и содержательной интерпретаций вводится обычным образом.

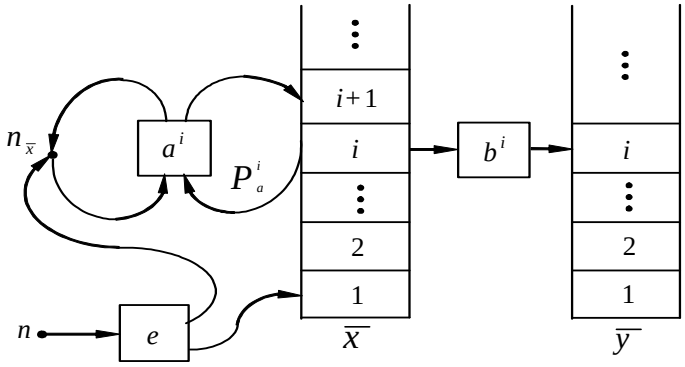


Рис.7.29

На рис.7.29. показана массовая операция a итеративной ВММ, $in(a^i)=\{ n_{\bar{x}}, \bar{x}\ [i]\}$, $out(a^i)=\{ n_{\bar{x}}, \bar{x}\ [i+1]\}$, $(\{n\},\{ \bar{x} \})$-план, вычисляющий массив $\bar{x}$ **,** содержит все термы вида $t_i=a^i(a^{i-1}(\ldots(a^1(\bar{x}\ [1])$. При должной интерпретации алгоритм, заданный $(\{n\},\{ \bar{x} \})$-планом, может считывать в оперативную память все записи файла.

На рис.30а показана структурированная итеративная ВММ C. Структурная интерпретация массовых операций a и b приведена на рис. 30б и 30в. Операциям a и b ВММ C (рис. 7.30а) соответствуют ВММ $C_a=SI(a)$ (см. рис. 7.30б) и $C_b=SI(b)$ (см. рис. 7.30в); $na=SI(n)$; переменные $SI(nx)$, $SI(ny)$, $SI(nz)$ — это счетчики $n_{\bar{x}}$, $n_{\bar{y}}$, $n_{\bar{z}}$, а $SI(x)$, $SI(y)$, $SI(z)$ — массивы $\bar{x}$, $\bar{y}$, $\bar{z}$ в ВММ C_a и C_b.

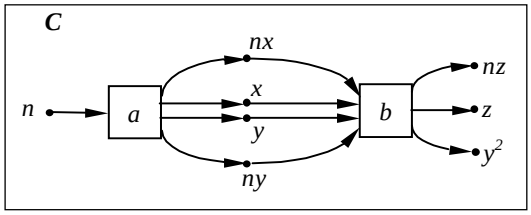


Рис. 7.30а.

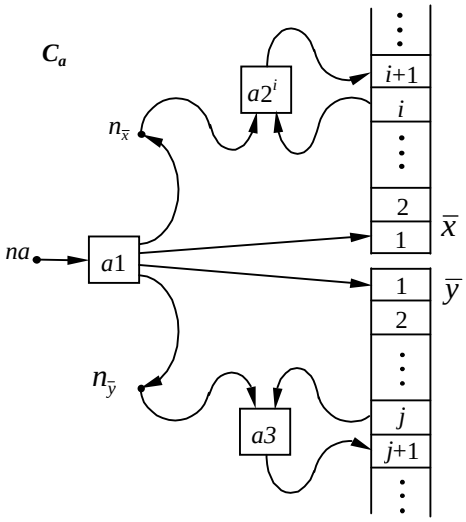


Рис. 7.30б

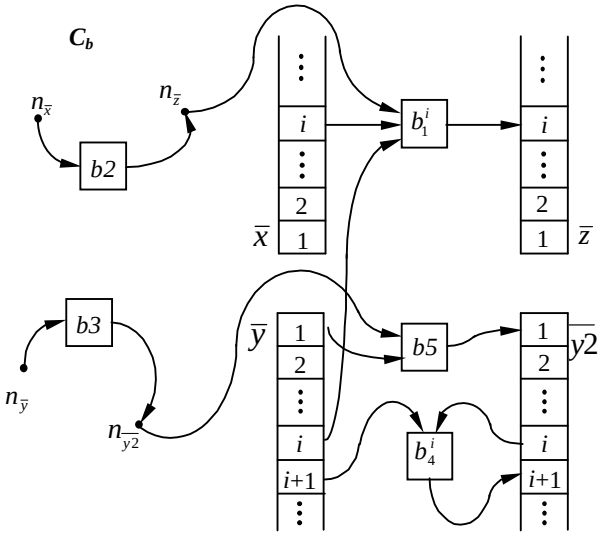


Рис. 7.30в

Если содержательно интерпретировать:

$I(n)$: (цельное положительное число),

$I(a1)$: $n_{\bar{x}} := n_{\bar{y}} := n$; ввод $\bar{x}$ [1], $\bar{y}$ [1])),

$I(a2^i)$: (ввод $\bar{x}$ [i + 1])), $I(a3^j)$: (ввод $\bar{y}$ [j + 1]),

$I(b2)$: ($n_{\bar{z}} := n_{\bar{x}}$), $I(b3)$: ($n_{\overline{y2}} := n_{\bar{y}}$),

$I(b1^i)$: ($\bar{z}$ [i]: = $\bar{x}$ [i] + $\bar{y}$ [i]),

$I(b4^i)$: ($\overline{y2}$ [i + 1]: = 1); печать ($\bar{y}$ [i + 1]),

$I(b5)$: (печать ($\bar{y}$ [i])),

область интерпретации — вещественные числа, тогда ({n},{nz,z,$y2$})-план задает алгоритм ввода и сложения массивов

$\overline{x}$ и $\overline{y}$ с результатом $\overline{z}$ и печатью $\overline{y}$, т. е. его термы вырабатывают значения компонентов $\overline{z}[i]$, равные сумме значений компонентов $\overline{x}[i]$ и $\overline{y}[i]$. Максимально непроцедурная форма задания этого алгоритма множеством термов $(\{n\},\{nz,z,y2\})$-плана позволяет (нет запрещающих ограничений) конструировать различные программы, реализующие его, например:

А. Сначала вводятся $\overline{x}$ и $\overline{y}$, а затем вычисляется $\overline{z}$ и печатается $\overline{y}$;

Б. Начинают вводиться $\overline{x}$ и $\overline{y}$, и по мере ввода вычисляться компоненты $\overline{z}$ и печататься компоненты $\overline{y}$;

В. Вводится $\overline{y}$ и в процессе ввода сразу печатаются его компоненты. По завершении ввода $\overline{y}$ начинает вводиться $\overline{x}$ и вычисляться $\overline{z}$ и, возможно, продолжают печататься компоненты $\overline{y}$;

Г. В целях экономии памяти массивам $\overline{x}$ и $\overline{y}$ отводятся ячейки для размещения только одного компонента. Тогда ввод осуществляется по одному компоненту $\overline{x}$ и $\overline{y}$, вычисляется соответствующий компонент $\overline{z}$ и печатается $\overline{y}$. Затем цикл повторяется.

Выбор конкретного вида программы определяется количеством ресурсов ЭВМ, выделенных для реализации

алгоритма, и функционалом, который характеризует качество программы.

7.3.4. Синтез максимально асинхронной программы

При рассмотрении вопросов конструирования максимально асинхронной программы следует иметь ввиду, что достаточно построить А-схему, реализующую потенциально бесконечный (*V*,*W*)-план. Рассмотрим основные требуемые идеи, понятия, конструкции и построения, необходимые для реализации такого (*V*,*W*)-плана.

Напомним, что А-схема состоит из конечного множества А-блоков $\{A_k \mid k \in \{1,2,...,l\}\}$, определенных над информационной *M* и управляющей *G* памятями, каждый из которых содержит спусковую функцию *tr*(*ak*), управляющий оператор *c*(*ak*) и операцию *ak*, $ak \in F_1$. Понятно, что каждая массовая операция должна реализоваться А-блоком, который должен срабатывать на всех входных наборах in(a^K) массовой операции a^K. Каждой переменной из $in(a^K) \cup out(a^K)$ в памяти *M* соответствует ячейка, в которой хранится значение. Для динамического массива $\overline{x}$ ячейки памяти отводятся в момент вычисления значения $n_{\overline{x}}$. Для компонентов итеративного массива $\overline{y}$ ячейки памяти заводятся в моменты изменения значений счетчика $n_{\overline{y}}$.

Выполнение А-блока A_k, реализующего массовую операцию, может быть инициировано в некоторый момент, если

tr(*ak*)=*true* при текущем состоянии памяти $M \cup G$. Спусковая функция tr(*ak*) вычисляется на всех входных наборах $in(a^к)$ таких, что для любого компонента массива $\overline{x}[j] \in in(a^к)$ выполняется $n_{\overline{x}} \geq j$ и блок *ak* запускается на счет с входными переменными $in(a^к)$ и выходными $out(a^к)$, т.е. реализуется операция $a^к$. В качестве упражнения можно построить детальный алгоритм синтеза максимально асинхронной программы, основываясь на предшествующих рассуждениях.

Алгоритмы планирования и оптимизирующих преобразований вычислений на моделях с массивами не обсуждаются ввиду их значительной сложности и меньшей важности для понимания проблем конструирования параллельных программ. Их детальное описание может быть найдено в [5].

7.3.7. Модификации и приложения метода

Метод синтеза параллельных программ на вычислительных моделях был рассмотрен в весьма общем виде. Для каждого конкретного приложения он должен быть модифицирован (конкретизован) для учета специфических особенностей предметной области с тем, чтобы более точно описывать алгоритмы и требуемые свойства конструируемых программ. Такая модификация обычно весьма не проста и делается она существенно по-разному для разных приложений. Это демонстрируют и приведенные ниже примеры различных приложений метода синтеза параллельных программ на вычислительных моделях. Конечно, по необходимости, эти приложения также описаны в довольно общем виде и для реализации требуют дальнейших значительных уточнений.

7.4.1. Интеллектуализация модульного программирования

Хорошо известно, что программное обеспечение является наиболее дорогим компонентов вычислительных систем. С другой стороны, для большинства задач ныне можно найти либо готовые решения, либо подходящие фрагменты, из которых можно собрать решение новой задачи (быть может, частично). По этой причине весьма злободневна проблема повторного использования накопленных программных модулей. Можно

решать эту задачу на пути построения интеллектуальных систем модульного программирования.

Одним из основных принципов интеллектуальных систем состоит в том, чтобы запоминать, накапливать в памяти ЭВМ знания о предметной области в активной форме, которая позволяет использовать эти знания при автоматическом либо человеко-машинном решении задач. Интеллектуальные системы накапливают как наиболее глубокие, специальные знания в предметной области (те самые, которые требуют высокого профессионализма от людей, работающих в предметной области), так и наиболее рутинные, “технические” знания, не знание которых людьми являются причиной огромного числа ошибок и с которыми человеку трудно справиться. Последнее весьма характерно для параллельного программирования, в котором сложилась ситуация, когда специалиста в предметной области должна окружать команда программистов для параллельной реализации его задач, что, безусловно, не нормально. Каждый человек должен уметь сам решать свои задачи на компьютере!

Систему синтеза параллельных программ (ССПП) можно рассматривать как интеллектуальную систему, которая кроме знаний о предметной области, об алгоритмах предметной области, способна еще накапливать и применять знания о методах и средствах конструирования параллельных программ

(ПП), о свойствах и правилах включения в ПП программных модулей, задающих содержательную интерпретацию операций вычислительных моделей. На этом и основана идея создания системы модульного программирования (СМП) на базе ССПП.

Применение ССПП для интеллектуализации модульного программирования позволяет решить ряд проблем повторного использования программных моделей.

1.Неактивное использование модулей из библиотеки модулей (БМ) СМП. Обусловлено это в первую очередь большой трудоемкостью включения модуля в многомодульную программу, так как для этого надо знать спецификации входных и выходных переменных модуля, требуемые для его исполнения ресурсы, правила обращения к модулю. Нужно также знать особенности реализованного в модуле алгоритма, уметь оценивать ошибки округления и т.д. и т.п. Именно эти знания и надо ввести в ССПП.

Трудно, кроме того, обеспечить поиск нужного модуля в БМ для вычисления требуемой функции *f*. Трудности эти еще увеличиваются, если надо найти не любой, а лучший модуль из имеющихся в БМ. Проблема здесь состоит не только и не столько в сложности поиска нужного модуля, сколько в сложности обеспечении автоматической замены модуля в готовой ПП, если в БМ появились лучшие модули либо при изменении в структуре мультикомпьютера. Например, при переносе ПП с одного

мультикомпьютера на другой может понадобиться замена модулей, которые просто не могут быть исполнены на оборудовании нового мультикомпьютера.

2.Ограниченный уровень модулей в БМ. Это положение иллюстрирует следующий пример. Существует много разных алгоритмов умножения матриц: универсальные, специальные (для жордановых, треугольных, *n*-диагональных и т.д.), которые реализуются разными модулями. Во всем этом многообразии необходимо ориентироваться при создании ПП. Выбор нужного модуля зависит также от мультикомпьютера, на котором должна выполняться ПП, нужно обеспечивать мобильность ПП в классе мультикомпьютеров, замену модулей при переносе ПП на другой мультикомпьютера. В этом случае желательно было бы ввести в БМ объемлющий модуль *УМ* (умножение матриц), обращение к которому открывает доступ ко всем имеющимся в БМ модулям умножения матриц. Прямо включать в ПП модуль *УМ* нельзя, это приведет к потере эффективности ПП, необходимо решить проблему выбора нужного модуля на этапе компиляции (а не на этапе выполнения) ПП. Здесь и может быть полезна ССПП, в которой интегральный модуль *УМ* может быть определен как структурированная операция и так включается в ПП. На этапе компиляции алгоритмы чистки структурированной операции, планирования и выбора обеспечат поиск нужного модуля,

удаление ненужных и корректное включение элементарного (не структурированного) модуля в ПП.

3.Большая трудоемкость включения модуля в БМ. Речь не идет конечно о простой записи имени нового модуля в каталог БМ. Для корректного использования в дальнейшем модуля в ПП, кроме полного знания о самом модуле (см.п.1), необходимо знать, как он соотносится с другими модулями уже имеющимися в БМ, провести сравнительный анализ свойств всех модулей БМ, определить случаи, когда предпочтительно использование нового, а когда старых модулей (переопределить для них область применение). Нужно также определить правильную передачу значений из/в нового модуля в старые. Следует также обеспечить поиск нового модуля в БМ.

Можно указать еще на следующие недостатки нынешних СМП:

1)надежность ПП оказывается ниже, чем можно получить на основе БМ.

2)качество ПП (по времени, требуемым ресурсам) также не лучшее, как правило, из-за того, что используются знакомые, хорошо изученные модули БМ, а не лучшие.

3)трудно построить ПП с заданными свойствами (степень надежности, время исполнения, класс используемых входных данных и т.д.).

Посмотрим теперь, как на основе вычислительных моделей может быть построена СМП. Основу ее составляет БМ, которую можно считать линейно упрядоченным списком модулей. Первая задача - превратить модули БМ в операции вычислительной модели и создать библиотеку операций. Операция вычислительной модели отличается от модуля, в частности, тем, что она всегда выполнима, если известны значения ее входных переменных. Свойство это обеспечивается тем, что употребление операции ВМ строго регламентировано - попросту точно указано, какие переменные следует взять в качестве входных и выходных и в каком конкретно алгоритме. Столь строго ограничить применение модуля в СМП, естественно, нельзя. Модули накапливаются в СМП как раз для реализации типичных алгоритмов при решении разнообразных задач, а не одной конкретной, как в вычислительных моделях. Следовательно, операции СМП должны содержать подробное описание всех тех свойств модуля, знание которых необходимо для его правильного включение в создаваемую ПП.

Для представления знаний о свойствах реализации конкретного модуля потребуется несколько расширить вычислительной модели, ввести понятие условной структурной интерпретации, с помощью которого можно будет определять сразу класс структурированных операций, и добавить управляющие переменные - одно из средств выбора

структурированных операции из заданного класса. При планировании управляющие переменные ничем не отличаются от прочих переменных модели, но в сконструированную ПП не попадают. Все рассмотрение ведется на примере.

Пусть необходимо ввести в СМП модуль *det*, для чего следует ввести его в состав структурированной операции *a* (рис. 7.31.*а*). Пусть модуль *det* вычисляет определитель *r* матрицы *m* размерности *n*×*n*. После исполнения *det* матрица *m* в памяти не сохраняется (модуль *det*, например, использует память, отведенную для размещения *m*, в ходе вычислений). Значения элементов *m* вещественные, имею формат одинарной точности и хранятся в памяти все, результат *r* также размещается в оперативной памяти.

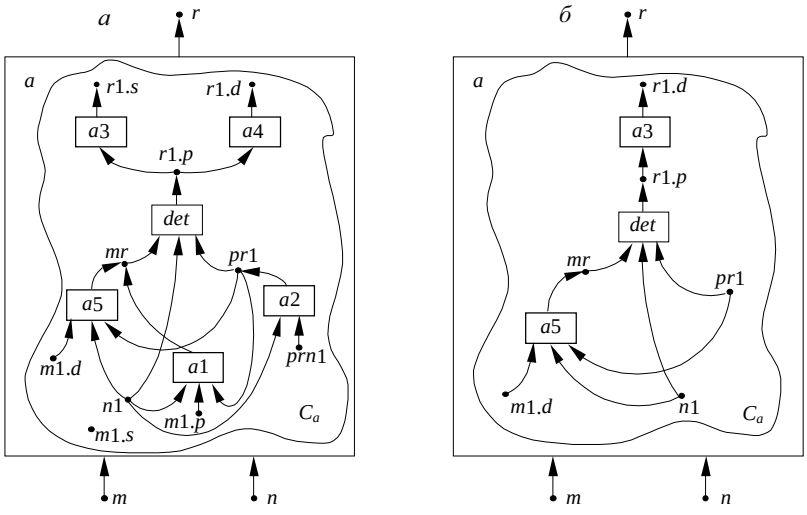


Рис. 7.31

Операция *a* (рис. 7.31.*a*) хранит эти сведения о свойствах модуля *det*. В C_a операции *a*1-*a*5 вычисляют следующие функции (имеются ввиду, конечно, функции *I*(*a*1)-*I*(*a*5)):

1) *a*1 копирует *m* в промежуточный массив *mr*, что позволяет сохранить *m* после выполнения *det*;

2) *a*2 проверяет свойства значения *n*;

3) *a*3 преобразует результат выполнения *det* в строку литер;

4) *a*4 преобразует результат в переменную двойной точности;

5) *a*5 преобразует значения элементов *m*, представленных в формате двойной точности, к одинарной, требуемой в *det*, $SI(a)=C_a$, $SI(m)=(m1.p|m1.d|m1.s|mr)$.

Здесь символ | означает “или”, т.е., переменной *m* в структурной интерпретации могут соответствовать либо *m*1.*p*, либо *m*1.*d*, либо *m*1.*s*, либо любая их комбинация.

Интеллектуализация СМП хороша еще и тем, что каждый разработчик модуля в состоянии во всех необходимых деталях описать правила его корректного использования в СМП, а пользоваться этим смогут все желающие.

7.4.2 Моделирование дискретных систем

На задаче моделирования дискретных систем рассмотрим возможности использования в этой ПО метода синтеза программ на вычислительных моделях и его необходимые модификации, учитывающие специфику ПО. Назовем дискретной системой (ДС) всякую систему, функционирование которой описывается алгоритмом, причем алгоритм этот в нашем случае задается множеством термов T_V^W , построенных на ВММ *C*.

Пусть задана ВММ *C*, каждой операции $a \in F_1$ сопоставляется функциональный символ $\bar{\tau}_a$, $T_1 = \bigcup_{a \in F_1} \{\bar{\tau}_a\}$. Каждому экземпляру a^i соответствует экземпляр $(\bar{\tau}_a, i) \in$ T, $\mathrm{T} = T_1 \times N; (\bar{\tau}_a, i)$ обозначается $\bar{\tau}_a^i$, $in(\bar{\tau}_a^i) = in(a)$. Набор *S=(C,T)* называется схемой ДС. Интерпретацией *S* в области *D* называется функция *I*, которая определяет корректную интерпретацию ВММ *C* и, кроме того, сопоставляет каждому функциональному символу $\bar{\tau}_a \in T$ вычислимую функцию $\tau_a : D^n \to R + \bigcup\{\Omega\}$, где $n = |in(a)|$, R^+ - множество положительных вещественных чисел; Ω - неопределенное значение. Функция τ_a для каждого такого набора значений входных переменных *in*(*a*), что предикат p_a принимает значение

true, определяет время вычисления функции $I(a)=f(a)$ (время выполнения операции a) на этих данных.

Набор $M=(S,I)$ называется имитационной моделью (ИМ) ДС. Каждой операции C соответствует функциональный элемент ДС, а каждой структурированной операции a, $SI(a)=C_a$, - некоторый функциональный блок ДС, схема которого $S_a=(C_a,T_a)$ называется подсхемой S, а $M_a=(S_a,I_a)$ - имитационной подмоделью M. Способ реализации термов T_V^W определяет тип моделирования ДС, а программа, реализующая термы нужным способом, называется моделью ДС. Для M можно выделить следующие три типа.

1. **Функциональное моделирование** ДС характеризуется тем, что время выполнения операции τ_a не учитывается. Операции срабатывают по мере вычисления значений всех своих входных переменных в произвольной последовательности. Функциональной моделью является любая ПП, реализующая T_V^W. Для упрощения можно синтезировать ПП, в которой структурированная операция выполняется только после вычисления всех ее аргументов. На этапе функционального моделирования выясняется, может ли ДС реализовать запланированные функции и частично проверяется корректность интерпретации. Этот тип моделирования целесообразно использовать на начальном этапе построения ИМ, так как он

требует значительно меньших вычислительных ресурсов, чем другие виды моделирования.

2. **Временно́е моделирование** осуществляется с учетом времени выполнения операции. Для реализации временно́й модели должна быть синтезирована ПП, в которой ведется время модели. Операция a может начать выполняться в момент τ_0, когда вычислены значения всех переменных из $in(a)$, а переменные из $out(a)$ получают значения лишь в момент $\tau_0 + \tau_a(in(a))$. Время выполнения подмодели определяется при выполнении. Как обычно, подмодель выполнена, когда завершаются все ее операции. Условия начала выполнения операции зависят от физических свойств элементов ДС, например, может требоваться, чтобы все значения входных переменных были вычислены в интервале $(\tau_0 - \varepsilon, \tau_0)$, $\varepsilon > 0$. Временно́е моделирование позволят выяснить время работы ИМ и ее подмоделей и обнаружить ошибки, связанные с несвоевременной выработкой значений переменных. Могут быть также обнаружены одинаковые операции, (их интерпретация задается одним и тем же модулем), и подсхемы, которые никогда не работают одновременно, и тогда, возможно, их функции удастся реализовать одним физическим устройством.

3. **Ресурсное моделирование** ДС – это временно́е моделирование в условиях, когда на ИМ дополнительно наложены ограничения на ресурсы. Может, например, требоваться, чтобы через контакт k передавались два разных сигнала x и y, либо чтобы некоторые подмодели $M1$ и $M2$ реализовывались в ДС одним и тем же физическим устройством, выполняющим в разное время функции $M1$ и $M2$. В синтезированной ПП, реализующей ресурсную модель, эти требования отражаются в распределении ресурсов вычислительной системы, и в устройстве прямого управления, в частности, для хранения значений переменных x и y должна быть отведена одна и та же ячейка, а подмоделям $M1$ и $M2$ должна соответствовать одна и та же процедура $П_{12}$. Во время выполнения ПП необходимо проводить контроль за правильным использованием значений x и y и выполнением $П_{12}$. При этом, например, значение x не должно использовать операции, которым необходимо значение y, а если процедура $П_{12}$ выполняется с входными значениями подмодели $M1$, то в это время не должны вырабатываться значения входных переменных подмодели $M2$.

Целью ресурсного моделирования является выявление ошибок, связанных с некорректным распределением ресурсов ДС, а также поиск оптимальных количественных характеристик ДС. В процесс проектирования вычислительных систем при

ресурсном моделировании может быть, например, выяснен вопрос, не приведет ли сведение каналов межпроцессорной коммутации в общую шину к недогрузки процессоров в том классе задач, для решения которых предназначается вычислительная система. При ресурсном моделировании ДС *АК* (асинхронный канал) может выясняться вопрос об оптимальном размере внутренней памяти *АК*. Схема *АК* приведена на рис. 7.32, массив $\bar{x}$ и $\bar{z}$ моделируют входной и выходной регистры канала, а массив $\overline{qu}$ - очередь (внутреннюю память); операция *a* определяет режим поступления данных на вход *АК*, а *b* - режим выборки данных из *АК*; операция *вз* считывает поступившие на входной регистр $\bar{x}$ данные и запоминает их во внутренней памяти *АК*; операция *выв* передает данные из памяти *АК* в выходной регистр $\bar{z}$, *i*-е компоненты массивов хранят *i*–е данное, передаваемое через *АК*. Задача определения оптимального размера очереди $\overline{qu}$ сводится в ресурсной модели к определению оптимального числа ячеек памяти МВС для хранения значений компонент $\overline{qu}$ в синтезированной программе.

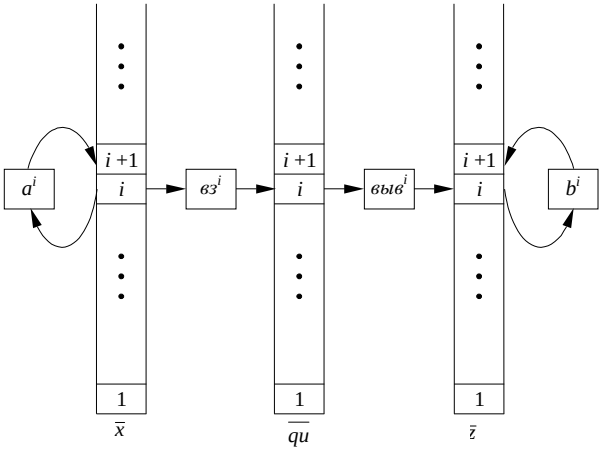


Рис. 7.32

Кроме выделенных типов моделирования можно определить и другие, например, тепловое моделирование, если сопоставить каждой операции функцию, значение которой задает количество выделившегося при выполнении функционального элемента ДС тепла.

Реализация системы автоматизации проектирования ДС (САПР) в виде ССПП открывает следующие пути улучшения качества проектируемых ДС, Прежде всего в ССПП допускаются различные преобразования (*V*,W)-плана, в частности, выделение и соединение подопераций структурированной операции, чему в ДС соответствует модификация ее структуры. Такие модификации делаются по директивам пользователя либо автоматически в

попытках найти оптимальную относительно какого-либо критерия структуру ДС. На улучшение рабочих характеристик САПР (время моделирования, требуемые ресурсы и т.п.) повлияет тот факт, что ССПП для каждой ДС синтезирует свою специальную ПП, реализующую тот или иной тип модели ДС, вместо универсальной программы интерпретирующего типа. В синтезированной ПП уже учтены особенности вычислительной системы, её ресурсы используются максимально при выполнении ПП, все изменения в структуре вычислительной системы будут также автоматически учитываться. Это обеспечивает переносимость САПР без потери производительности при разумных ограничениях. Выделение типов моделирования позволяет поэтапно отлаживать ИМ, выявляя последовательно логические, временные и ресурсные ошибки, а это ускорит отладку ИМ. И, наконец, ССПП обеспечивает функциональное проектирование ДС, при котором ДС конструируется из функциональных блоков, внутреннее устройство которых задается структурной интерпретацией. При этом каждому функциональному блоку (структурированной операции ВММ) *a* соответствует ВМ C_a, определяющая всевозможные алгоритмы

вычисления f_a, из которых при раскрытии a выбирается с использованием «подсказок» пользователя лучший.

XIII. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В СИСТЕМАХ MPI и OpenMP.

Вычислительные системы с распределенной и общей памятью являются доминирующими на рынке суперкомпьютеров. Самым распространенными системами параллельного программирования, ориентированными на эти вычислительные системы, являются *MPI* (*Message Passing Interface*) [20] и *OpenMP* [21]. На данный момент это наиболее развитые системы параллельного программирования.

В главе приведены примеры параллельного программирования на системах обоих типов. Рассматриваются примеры параллельного программирования алгоритмов решения следующих задач: умножения матрицы на матрицу, решение систем линейных уравнений методом Гаусса и итерационными методами. Причем для сравнения алгоритм каждой задачи запрограммирован как в *MPI* так и в *OpenMP*. На этих примерах видна разница в программировании на распределенных системах и системах с общей памятью.

Здесь не приводятся временные характеристики рассматриваемых алгоритмов. Точные замеры времени решения конкретного параллельного алгоритма могут быть сделаны на конкретной вычислительной системе на некотором наборе *данных*. Ускорение параллельных алгоритмов зависит, во-первых, от вычислительной системы, на которой выполняется задача, а, во-вторых, от структуры самих алгоритмов. Получение конкретных характеристик алгоритмов, например, таких как ускорение и эффективность, является предметом исследования на лабораторных занятиях.

8.1. Введение

8.1.1. Модели параллельного программирования

Модели параллельного программирования рассматриваются с разных точек зрения:

1) С точки зрения доступа к разделяемым данным и синхронизации процессов. В настоящее время в области научно-

технических вычислений превалируют три модели параллельного программирования: модель передачи сообщений (МПС), модель с общей памятью (МОП) и модель параллелизма по данным (МПД).

2) С точки зрения построения ветвей параллельного алгоритма.

MPMD - модель вычислений (*Multiple program - Multiple Data*). Параллельная программа представляет собой совокупность автономных процессов, функционирующих под управлением своих собственных программ и взаимодействующих посредством стандартного набора библиотечных процедур для передачи и приема сообщений.

SPMD - модель вычислений (*Single program - Multiple Data*). Все процессы исполняют в общем случае различные ветви одной и той же программы. Такой подход обусловлен тем обстоятельством, что задача может быть достаточно естественным образом разбита на подзадачи, решаемые по одному алгоритму. Кратко, суть этого способа заключается в следующем. Исходные данные задачи распределяются по процессам (ветвям параллельного алгоритма), а алгоритм является одним и тем же во всех процессах, но действия этого алгоритма распределяются в соответствии с имеющимися в этих процессах *данными*. Распределение действий алгоритма заключается, например, в присвоении разных значений переменным одних и тех же циклов в разных ветвях, либо в исполнении в разных ветвях разного количества итераций одних и тех же циклов и т.п. Другими словами, процесс в каждой ветви следует различными путями выполнения на той же самой программе. На практике чаще всего встречается именно эта модель программирования.

3) С точки зрения размеров (по объему и времени) обрабатываемых параллельных блоков.
Это методы крупнозернистого, среднезернистого и мелкозернистого распараллеливания.

8.1.2.Модель передачи сообщений [22,23]**.** В модели передачи сообщений каждый процесс имеет собственное

локальное адресное пространство. Обработка общих данных и синхронизация осуществляется посредством передачи сообщений.

В рамках данной модели параллельного программирования реализуются две подмодели с точки зрения построения ветвей параллельной программы. *MPMD*-модель вычислений и *SPMD*-модель вычислений.

Для параллельных систем с передачей сообщений оптимальное соотношение между вычислениями и коммуникациями обеспечивают методы **крупнозернистого** распараллеливания, когда параллельные алгоритмы строятся из крупных и редко взаимодействующих блоков. Многие задачи линейной алгебры, задачи, решаемые сеточными методами и многие другие, достаточно эффективно распараллеливаются крупнозернистыми методами.

Самой распространенной системой параллельного программирования, поддерживающим данную модель, является система *MPI*.

8.1.3.Модель с общей памятью [23]**.** В модели с общей памятью процессы разделяют общее адресное пространство. Так как нет ограничений на использование общих данных, то программист должен явно специфицировать общие данные и упорядочивать доступ к ним с помощью средств синхронизации.

Поскольку в этой модели нет коммуникаций, как в предыдущей модели передачи сообщений, сдерживающих скорость обработки общих данных, то эта модель в большей степени относится к **среднезернистой** модели программирования. То есть в этой модели параллельные процессы могут строиться из небольших блоков. Причем параллельные процессы, обрабатывающие такие блоки, как правило, достаточно часто инициируются и завершаются в процессе вычислений.

В рамках данной модели параллельного программирования реализуются, как и в предыдущем случае, две подмодели с точки зрения построения ветвей параллельной программы: *MPMD*-модель вычислений и

SPMD-модель вычислений. По одному и тому же алгоритму в разных ветвях обрабатываются разные данные. Например, при распараллеливании циклов, разные ветви выполняют одни и те же циклы (при разных значениях переменных), но обрабатывают разные данные (*MPMD*-модель вычислений). В то же время возможно параллельное выполнение некоторых участков программы с разными кодами, т.е. использует *MPMD*-модель вычислений.

Обобщение и стандартизация моделей с общей памятью привели к созданию стандарта системы параллельного программирования *OpenMP*.

Рассмотренные выше модели МПС и МОП, помимо поддержки на языковом уровне, поддерживаются архитектурами таких самых современных суперкомпьютеров как: *ASCI RED* (более 9000 *Pentium PRO*/200 объединены в единую систему, имеет быстродействие около 3,2 *Tflops*), и *ASCI WAIT* (8192 - , имеет быстродействие 12,2 *Tflops*), *Cray T3D, Cray T3E, IBM SP2* и многими другими.

8.1.4.Модель параллелизма по данным. В модели параллелизма по данным отсутствует понятие процесса и, как следствие, явная передача сообщений или явная синхронизация. В этой модели данные последовательной программы распределяются по узлам (процессорам) вычислительной системы. Последовательная программа преобразуется компилятором либо в модель передачи сообщений, либо в модель с общей памятью (рис.8.1). При этом вычисления распределяются по правилу собственных вычислений: каждый процессор выполняет только вычисления собственных данных, т.е. данных, распределенных на этот процессор.

В рамках данной модели параллельного программирования реализуются две подмодели с точки зрения построения ветвей параллельной программы. *MPMD*-модель вычислений и *SPMD*-модель вычислений.

Эта модель реализуется методом **крупнозернистого** распараллеливания.

Модель освобождает программиста от рутинной и трудоемкой работы по распределению глобальных массивов на локальные массивы процессов, по управлению передачей сообщений и синхронизации доступа к общим данным. Однако применение этой модели является еще предметом исследований. Результаты исследований показывают, что эффективность некоторых параллельных алгоритмов в модели МПД сравнима с эффективностью реализации в моделях МПС и МОП.

8.1.5. В чем особенность и сложность параллельного программирования. Чтобы представить себе трудности параллельного программирования нужно ответить на вопрос: в чем одно из важных отличий в написании последовательной и параллельной программ? Если это параллельная программа для систем с передачей сообщений, то прежде чем создавать такую параллельную программу, необходимо знать общую архитектуру параллельной машины и топологию межпроцессорных связей, которая существенно используется при программировании. Это связано с тем, что невозможно создание средства автоматического распараллеливания, которое позволяло бы превращать последовательную программу в параллельную, и обеспечивало бы ее высокую производительность. Поэтому в параллельной программе приходится в явном виде задавать операторы создания топологий коммуникационной сети и операторы обменов данными между процессорами, что приводит к снижению уровня программирования. Кроме того, пользователю нужно обеспечивать правильность взаимодействий множества параллельно выполняющихся независимо друг от друга процессов, что затрудняет отладку программы. При написании же последовательной программы знать архитектуру процессора, на котором будет исполняться

программа, зачастую нет необходимости, поскольку учет особенностей архитектуры скалярного процессора может быть сделан компилятором с приемлемыми потерями в производительности программы.

Если речь идет о параллельной программе для систем над общим полем памяти, то программист должен явно специфицировать общие данные и упорядочить доступ к ним с помощью средств синхронизации, что требует дополнительных усилий и умения работать, например, с семафорами. Это приводит, как и в предыдущем случае, снижению уровня программирования и часто к трудностям при отладке программы.

В модели параллелизма по данным отсутствует понятие процесса и, как следствие, явная передача сообщений или явная синхронизация. И это, по сравнению с двумя предыдущими моделями, повышает уровень программирования. Но и в этой модели, по-прежнему, нужно явно задавать множество процессоров мультикомпьютера и его структуру связей и явно распределять данные программы по узлам (процессорам) вычислительной системы. Последовательная программа затем преобразуется компилятором либо в модель передачи сообщений, либо в модель с общей памятью. При этом каждый процессор выполняет только вычисления собственных данных, т.е. данных, распределенных на этот процессор. И здесь, как правило, появляется много "подводных камней", о которых трудно порой догадаться, т.к. отображение алгоритма и данных на узлы вычислительной системы скрыты от пользователя, что затрудняет проверку качества такого отображения.

8.1.6. Почему выбраны *MPI, OpenMP* и *HPF* ? На данный момент это наиболее развитые системы параллельного программирования. Как и всякие системы, они постоянно развиваются и расширяются. *MPI* является сейчас самой развитой системой параллельного программирования с передачей

сообщений. Наиболее важным свойством *MPI* является то, что он позволяет создавать переносимые параллельные программы, достаточно эффективные и надежные.

Переносимость обеспечивается:

1) такими механизмами, как: определение *виртуального компьютера* (программно реализуемого) и возможностью задания произвольного количества таких виртуальных компьютеров в системе независимо от количества физических компьютеров.

2) заданием *виртуальных топологий* (программно реализуемых). Отображение виртуальных коммуникационных топологий на физическую осуществляется системой *MPI*. Виртуальные топологии обеспечивают оптимальное приближение архитектуры системы к структурам задач при хорошей переносимости задач.

3) тем, что тот же самый исходный текст параллельной программы на *MPI* может быть выполнен на ряде машин (некоторая настройка необходима, чтобы взять преимущество из элементов каждой системы). Программный код может одинаково эффективно выполняться, как на параллельных компьютерах с распределенной памятью, так и на параллельных компьютерах с общей памятью. Он может выполняться на сети рабочих станций, или на наборе процессоров на отдельной рабочей станции.

4) способностью параллельных программ выполняться на гетерогенных системах, то есть на системах, состоящих из процессоров с различной архитектурой. *MPI* обеспечивает вычислительную модель, которая скрывает много архитектурных различий в работе процессоров. *MPI* автоматически делает любое необходимое преобразование данных и использует правильный протокол связи, посылается ли код

сообщения между одинаковыми процессорами или между процессорами с различной архитектурой. *MPI* может настраиваться как на работу на однородной, так и на работу на гетерогенной системах.

5) компиляторами для *Fortran(a)* и *C*.

Эффективность и надежность обеспечиваются:

1) определением *MPI* операций не процедурно, а логически, т.е. внутренние механизмы выполнения операций скрыты от пользователя;
2) использованием непрозрачных объектов в *MPI* (*группы*, *коммуникаторы*, *типы* и т.д.);
3) хорошей реализацией функций передачи данных, адаптирующихся к структуре физической системы.

Обменные функции разработаны с учетом архитектуры системы, например, для систем с распределенной памятью, систем с общей памятью, и некоторых других систем, что позволяет минимизировать время обмена данными.

Почему *OpenMP*? *OpenMP* является стандартом для программирования на масштабируемых *SMP*-системах (*SSMP*, *ccNUMA*, и т.д.) в модели общей памяти (*shared memory model*). Примерами систем с общей памятью, масштабируемых до большого числа процессоров, могут служить суперкомпьютеры *Cray Origin2000* (до 128 процессоров), *HP* 9000 *V-class* (до 32 процессоров в одном узле, а в конфигурации из 4 узлов - до 128 процессоров), *Sun Starfire* (до 64 процессоров).

MPI тоже работает на таких системах, однако модель передачи сообщений недостаточно эффективна на *SMP*-системах. *POSIX*-интерфейс для организации нитей (*Pthreads*) поддерживается широко (практически на всех *UNIX*-системах), однако по многим причинам не подходит для практического параллельного программирования: нет поддержки *Fortran*-а, слишком низкий уровень программирования, нет поддержки параллелизма по данным.

OpenMP можно рассматривать как высокоуровневую надстройку над *Pthreads* (или аналогичными библиотеками нитей). Многие поставщики *SMP*-архитектур (*Sun, HP, SGI*) в своих компиляторах поддерживают специальные директивы для распараллеливания циклов. Однако эти наборы директив, как правило, 1) весьма ограничены; 2) несовместимы между собой; в результате чего разработчикам приходится распараллеливать приложение отдельно для каждой платформы. *OpenMP* является во многом обобщением и расширением упомянутых наборов директив.

Какие преимущества *OpenMP* дает разработчику?

1. За счет идеи "инкрементального распараллеливания" *OpenMP* идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст программы *OpenMP*-директивы.
2. При этом, *OpenMP* - достаточно гибкий механизм, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения.
3. Предполагается, что *OpenMP*-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы *OpenMP* просто игнорируются последовательным компилятором.
4. Одним из достоинств *OpenMP* его разработчики считают поддержку так называемых "*orphan*" (оторванных) директив, то есть директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области.

Способ распараллеливания на *HPF* принципиально отличается от двух предыдущих тем, что эта модель освобождает программиста от рутинной и трудоемкой работы по распределению глобальных массивов на локальные массивы процессов, по управлению передачей сообщений и синхронизации доступа к общим данным. И

этим эта модель интересна. Кроме того, эта система программирования имеет большую поддержку многих крупных фирм производителей.

Первичные цели при развитии *HPF* включают:

1. Поддержку параллелизма по данным.

2. Мобильность задания пересекающихся различных архитектур (в основном решеток).

3. Высокую эффективность на параллельных компьютерах с неоднородными затратами доступа к памяти.

4. Возможность использования *HPF*-программы на однопроцессорной платформе в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы *HPF* просто игнорируются последовательным компилятором.

5. Использование стандартного ФОРТРАНа (в настоящее время ФОРТРАН 95) как основы;

6. Открытые интерфейсы и функциональная совместимость с другими языками (например, *C*) и другим парадигмами (например, передача сообщений, используя MPI).

Вторичные цели включают:

1. Возможность расширения в случае расширения в будущим стандартов для ФОРТРАНа и *C*;

2. Эволюционный путь развития для того, чтобы была возможность непротиворечивым образом добавлять расширения к системе программирования.

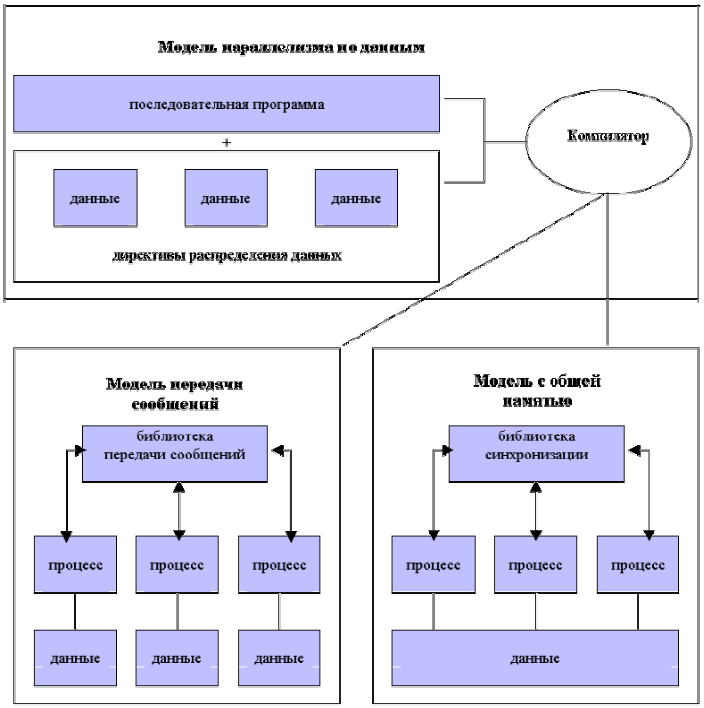


Рис.8.1. Три модели параллельного программирования

8.2.Программирование на распределенных мультикомпьютерах и примеры параллельных программ в MPI

В этом параграфе приводятся примеры параллельных алгоритмов решения задач на вычислительных системах с **распределенной памятью** с использованием системы параллельного программирования *MPI*. Приведенные здесь параллельные программы решения задач являются иллюстрационными, демонстрирующими способы распараллеливания задач подобного класса, и показывающими применение и возможности функций *MPI*. Информация по *MPI* может быть получена, например, в [20] или в информационных ресурсах сети Интернет.

MPI имеет явную (не автоматическую) модель программирования, предлагая программисту полное управление по распараллеливанию. *MPI* ориентирован на реализацию алгоритмов, *распараллеливаемых, в основном, крупнозернистыми методами*. Для представления алгоритмов используется, в основном, *SPMD*-модель вычислений (*распараллеливание по данным*). *MPMD*-модель используется не так часто. Для реализации этой модели необходимо коды разных программ скомпоновать в одном исполняемом файле и при выполнении каждая ветвь будет выполнять код своей программы. В случае с *MPMD*-моделью более часто используется ситуация, когда параллельные ветви исполняют код разных участков одной и той же программы.

Представленные здесь примеры используют в основном *SPMD*-модель вычислений. Однородное распределение данных по компьютерам – основа для хорошего баланса времени, затрачиваемого на вычисления, и времени, затрачиваемого на взаимодействия ветвей параллельной программы. При таком распределении преследуется цель – равенство объёмов распределяемых частей данных и соответствие нумерации распределяемых частей данных нумерации компьютеров в системе. Исходными данными рассматриваемых здесь алгоритмов являются матрицы, векторы и 2*D* (двумерное) пространство вычислений. В этих алгоритмах применяются

следующие способы однородного распределения данных: *горизонтальными полосами*, *вертикальными полосами* и *циклическими горизонтальными полосами*. При распределении *горизонтальными полосами* матрица, вектор или 2*D* пространство "разрезается" на полосы по строкам (далее слово "разрезанная" будем писать без кавычек и матрицу, вектор или 2*D* пространство обозначать для краткости словом - *данные*). Пусть M – количество строк матрицы, количество элементов вектора или количество строк узлов 2D пространства, P – количество виртуальных компьютеров в системе, $C_1 = M/P$ – целая часть от деления, $C_2 = M\%P$ – дробная часть. *Данные* разрезаются на P полос. Первые $(P–C_2)$ полос имеют по C_1 строки, а остальные C_2 полосы имеют по C_1+1 строки. Полосы *данных* распределяются по компьютерам следующим образом. Первая полоса помещается в компьютер с номером 0, вторая полоса – в компьютер 1, и т. д. Такое распределение полос по компьютерам учитывается в параллельном алгоритме. Распределение *вертикальными полосами* аналогично предыдущему, только в распределении участвуют столбцы матрицы или столбцы узлов 2*D* пространства. И, наконец, распределение *циклическими горизонтальными полосами*. При таком распределении *данные* разрезаются на количество полос значительно большее, чем количество компьютеров. И чаще всего полоса состоит из одной строки. Первая полоса загружается в компьютер 0, вторая – в компьютер 1, и т.д., затем, (P-1)-я полоса снова в компьютер 0, P-я полоса в компьютер 1, и т.д.

Приведенные два алгоритма решения СЛАУ методом Гаусса показывают, что однородность распределения данных сама по себе еще недостаточна для эффективности алгоритма. Эффективность алгоритмов зависит еще и от способа распределения *данных*. Разные способы представления *данных* влекут, соответственно, и разные организации алгоритмов, обрабатывающих их.

8.2.1 Умножение матрицы на матрицу. Умножение матрицы на вектор и матрицы на матрицу являются базовыми макрооперациями для многих задач линейной алгебры, например

итерационных методов решения систем линейных уравнений и т. п. Поэтому приведенные алгоритмы можно рассматривать как фрагменты в алгоритмах этих методов. В этом пункте приведено три алгоритма умножения матрицы на матрицу. Разнообразие вариантов алгоритмов проистекает от разнообразия вычислительных систем и размеров задач. Рассматриваются и разные варианты загрузки *данных* в систему: загрузка данных через один компьютер; и загрузка данных непосредственно каждым компьютером с дисковой памяти. Если загрузка *данных* осуществляется через один компьютер, то *данные* считываются этим компьютером с дисковой памяти, разрезаются и части рассылаются по остальным компьютерам. Но *данные* могут быть подготовлены и заранее, т.е. заранее разрезаны по частям и каждая часть записана на диск в виде отдельного файла со своим именем; затем каждый компьютер непосредственно считывает с диска, предназначенный для него файл.

8.2.1.1 Алгоритм 1. Заданы две исходные матрицы A и B. Вычисляется произведение $C = A \times B$, где A - матрица $n_1 \times n_2$, и B - матрица $n_2 \times n_3$. Матрица результатов C имеет размер $n_1 \times n_3$. Исходные матрицы предварительно разрезаны на полосы, полосы записаны на дисковую память отдельными файлами со своими именами и доступны всем компьютерам. Матрица результатов возвращается в нулевой процесс.

Реализация алгоритма выполняется на кольце из p_1 компьютеров. Матрицы разрезаны как показано на рис. 8.2. матрица A разрезана на p_1 горизонтальных полос, матрица B разрезана на p_1 вертикальных полос, и матрица результата C разрезана на p_1 полосы. Здесь предполагается, что в память каждого компьютера загружается и может находиться только одна полоса матрицы A и одна полоса матрицы B.

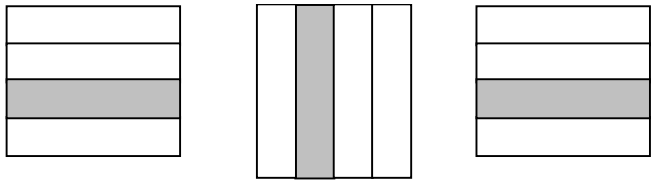


Рис. 8.2. Разрезание данных для параллельного алгоритма произведения двух матриц при вычислении на кольце компьютеров.

Поскольку по условию в компьютерах находится по одной полосе матриц, то полосы матрицы *B* (либо полосы матрицы *A*) необходимо "прокрутить" по кольцу компьютеров мимо полос матрицы *A* (матрицы *B*). Каждый сдвиг полос вдоль кольца и соответствующее умножение представлено на рис. 8.3 в виде отдельного шага. На каждом таком шаге вычисляется только часть полосы. Процесс `i` вычисляет на `j`-м шаге произведение `i`-й горизонтальной полосы матрицы *A* и `j`-й вертикальной полосы матрицы `B`, произведение получено в подматрице `(i,j)` матрицы *C*. Текст программы, реализующий алгоритм, приведен ниже.

Последовательные стадии вычислений иллюстрируются на рис. 8.3.

1. Каждый компьютер считывает с диска соответствующую ему полосу матрицы *A*. Нулевая полоса должна считываться нулевым компьютером, первая полоса - первым компьютером и т.д., последняя полоса - считывается последним компьютером. На рис. 8.3. полосы матрицы *A* и *B* пронумерованы.

2. Каждый компьютер считывает с диска соответствующую ему полосу матрицы *B*. В данном случае нулевая полоса должна считываться нулевым компьютером, первая полоса -

первым компьютером и т.д., последняя полоса - считывается последним компьютером.

3. Вычислительный шаг 1. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров.

4. Вычислительный шаг 2. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров. И т. д.

5. Вычислительный шаг p_1-1. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров.

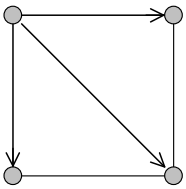
6. Вычислительный шаг p_1. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров.

7. Матрица C собирается в нулевом компьютере.

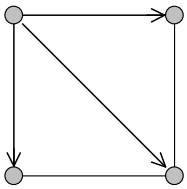
Если "прокручивать" вертикальные полосы матрицы B, то матрица C будет распределена горизонтальными полосами, а если "прокручивать" горизонтальные полосы матрицы A, то матрица C будет распределена вертикальными полосами. Алгоритм характерен тем, что после каждого шага вычислений осуществляется обмен *данными*.

Если, время передачи *данных* велико по сравнению с временем вычислений, либо каналы передачи *данных* медленные, то ускорение алгоритма, по сравнению с последовательным, будет не высока.

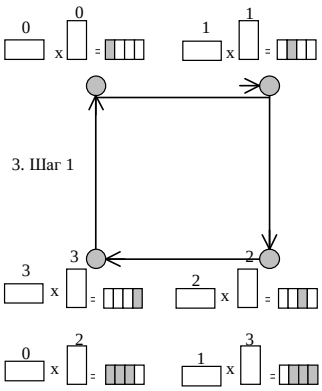
При достаточных ресурсах памяти в системе, конечно же, лучше использовать алгоритм, в котором минимизированы обмены между компьютерами в процессе вычислений. Это достигается за счет дублирования некоторых *данных* в памяти компьютеров. В следующих двух алгоритмах используется этот подход.

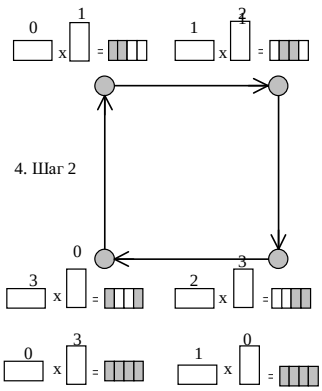


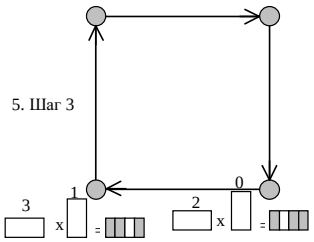
1. scatter A

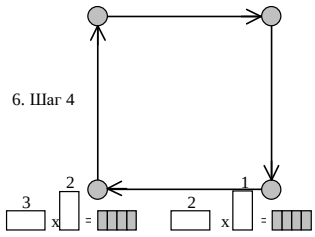


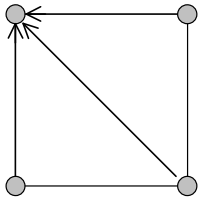
2. scatter B











7. Сбор результатов в C

Рис. 8.3. Стадии вычислений произведения матриц на кольце компьютеров.

```
/* Пример программы на языке C (с MPI)
 * произведения двух матриц в топологии "кольцо"
 * Здесь в примере исходные матрицы генерируются
 * в программе.
 * В примере предполагается, что количество
 * строк матрицы A и количество столбцов матрицы
 * B делятся без остатка на количество
 * компьютеров в системе.
 * В данном случае задачу запускаем на восьми
 * компьютерах.
 */
   #include<stdio.h>
   #include<mpi.h>
   #include<time.h>
   #include<sys/time.h>

/* Задаем в каждой ветви размеры полос матриц A,
 * B и C. (Здесь предполагается, что размеры
 * ветвей одинаковы во всех ветвях. */

   #define M 320
   #define N 40

/* NUM_DIMS - размер декартовой топологии.
 * "кольцо" - одномерный тор. */

   #define NUM_DIMS 1
   #define EL(x) (sizeof(x) / sizeof(x[0][0]))

/* Задаем полосы исходных матриц. В каждой
 * ветви, в данном случае, они одинаковы */

static double A[N][M], B[M][N], C[N][M];

int main(int argc, char **argv)
   { int rank,size,i,j,k,i1,j1,d,sour,dest;
     int dims[NUM_DIMS], periods[NUM_DIMS];
     int new_coords[NUM_DIMS];
     int reorder = 0;
     MPI_Comm   comm_cart;
     MPI_Status st;
```

```
    struct timeval tv1, tv2; /* Для засечения
                                времени */
    int dt1;
/* Инициализация библиотеки MPI*/
    MPI_Init(&argc, &argv);
/* Каждая ветвь узнает количество задач в
 * стартовавшем приложении */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* и свой собств. номер: от 0 до (size-1) */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Обнуляем массив dims и заполняем массив
   * periods для топологии "кольцо" */
    for(i=0; i<NUM_DIMS; i++)
     { dims[i] = 0;
       periods[i] = 1;
     }
/* Заполняем массив dims, где указываются
 * размеры (одномерной) решетки */
    MPI_Dims_create(size, NUM_DIMS, dims);
/* Создаем топологию "кольцо" с  comm_cart */
    MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS,
         dims, periods, reorder, &comm_cart);
/* Отображаем ранги на координаты компьютеров,
 * с целью оптимизации отображения заданной
 * виртуальной топологии на физическую
 * топологию системы. */
    MPI_Cart_coords(comm_cart, rank, NUM_DIMS,
                                 new_coords);
/* Каждая ветвь находит своих соседей вдоль
 * кольца, в направлении меньших значений
 * рангов */
    MPI_Cart_shift(comm_cart, 0, -1, &sour,
```

```
                                        &dest);
/* Каждая ветвь генерирует полосы исходных
 * матриц A и B, полосы C обнуляет */
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { A[i][j] = 3.141528;
            B[j][i] = 2.812;
            C[i][j] = 0.0;
          }
      }
 /* Засекаем начало умножения матриц */
    gettimeofday(&tv1, (struct timezone*)0);
  /* Каждая ветвь производит умножение своих полос
   * матриц. Самый внешний цикл for(k) - цикл
   * по компьютерам */
    for(k = 0; k < size; k++)
      {
    /* Каждая ветвь вычисляет координаты (вдоль
     * строки) для результирующих элементов
     * матрицы C, которые зависят от номера
     * цикла k и ранга компьютера. */
       d = ((rank + k)%size)*N;
    /* Каждая ветвь производит умножение своей
     * полосы матрицы A на текущую полосу матрицы
     * B */
       for(j = 0; j < N; j++)
         { for(i1=0, j1=d; j1 < d+N; j1++, i1++)
             { for(i = 0; i < M; i++)
                 C[j][j1] += A[j][i] * B[i][i1];
             }
         }
    /* Умножение полосы строк матрицы A на полосу
     * столбцов матрицы B в каждой ветви завершено
     */
    /* Каждая ветвь передает своим соседним ветвям
```

```
    * с меньшим рангом вертикальные полосы
    * матрицы B. Т.е. полосы матрицы B сдвигаются
    * вдоль кольца компьютеров */
      MPI_Sendrecv_replace(B,EL(B),MPI_DOUBLE,
                dest,12,sour,12,comm_cart, &st);
    }
/* Умножение завершено. Каждая ветвь умножила
 * свою полосу строк матрицы A
 * на все полосы столбцов матрицы B.
 * Засекаем время и результат печатаем */
   gettimeofday(&tv2, (struct timezone*)0);
   dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
                     tv2.tv_usec - tv1.tv_usec;
   printf("rank = %d Time = %d\n", rank, dt1);
/* Для контроля печатаем первые четыре элемента
 * первой строки результата */
   if(rank == 0)
     { for(i = 0; i < 1; i++)
         for(j = 0; j < 4; j++)
             printf("C[i][j] = %f\n", C[i][j]);
     }
/* Все ветви завершают системные процессы,
 * с топологией comm_cart и завершают
 * выполнение программы */
   MPI_Comm_free(&comm_cart);
   MPI_Finalize();
   return(0);
 }
```

8.2.1.2 Алгоритм 2. Вычисляется произведение $C = A \times B$, где A - матрица $n_1 \times n_2$, и B - матрица $n_2 \times n_3$. Матрица результатов C имеет размер $n_1 \times n_3$. Исходные матрицы первоначально доступны на нулевом процессе, и матрица результатов возвращена в нулевой процесс.

Параллельное выполнение алгоритма осуществляется на двумерной (2D) решетке компьютеров размером $p_1 \times p_2$.

Матрицы разрезаны, как показано на рис. 8.4: матрица A разрезана на p_1 горизонтальных полос, матрица B разрезана на p_2 вертикальных полос, и матрица результата C разрезана на $p_1 \times p_2$ подматрицы (или субматрицы).

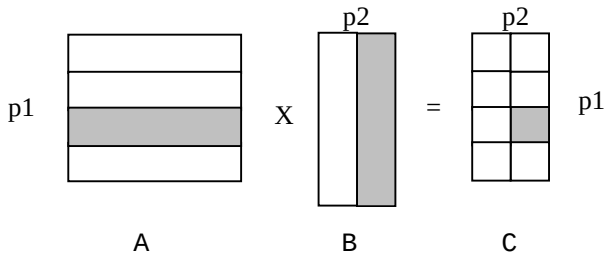


Рис. 8.4. Разрезание данных для параллельного алгоритма произведения двух матриц при вычислении в 2D решетке компьютеров. Выделенные данные расположены в одном компьютере

Каждый компьютер (i,j) вычисляет произведение i-й горизонтальной полосы матрицы A и j-й вертикальной полосы матрицы B, произведение получено в подматрице (i,j) матрицы C.

Последовательные стадии вычисления иллюстрируются на рис. 8.5:

1. Матрица A распределяется по горизонтальным полосам вдоль координаты $(x,0)$.

2. Матрица B распределяется по вертикальным полосам вдоль координаты $(0,y)$.

3. Полосы A распространяются в измерении y.

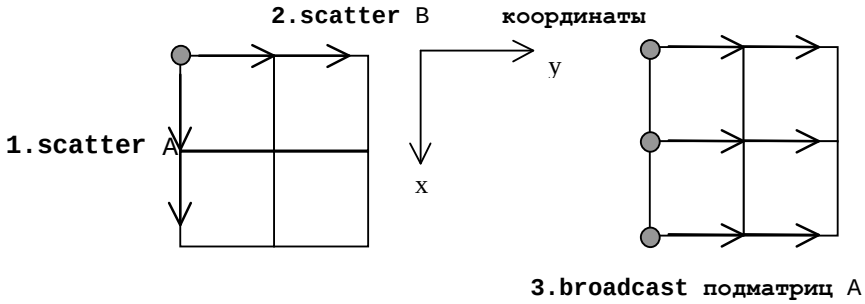
4. Полосы B распространяются в измерении x.

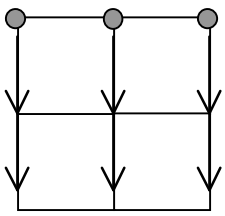
5. Каждый процесс вычисляет одну подматрицу произведения.

7. Матрица C собирается из (x, y) плоскости.

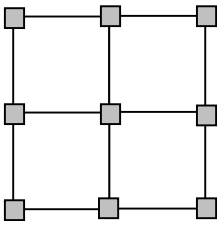
Осуществлять пересылки между компьютерами во время вычислений не нужно, т.к. все полосы матрицы A пересекаются со всеми полосами матрицы B в памяти компьютеров системы.

Этот алгоритм имеет большую скорость чем предыдущий, т.к. непроизводительное время пересылок данных осуществляется только при загрузке исходных данных в память компьютеров и их выгрузке, и нет обменов *данными* в процессе вычислений.

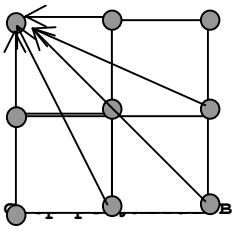




4.broadcast подматриц B



5.вычисление подматриц в C



6. в C

Рис. 8.5. Стадии вычисления произведения матриц в 2*D* параллельном алгоритме

```
/* Произведение двух матриц в топологии
 * "двумерная решетка". В примере, данные
 * генерируются в нулевом компьютере.
 * В примере предполагается, что количество строк
 * матрицы A и количество столбцов матрицы B
 * делятся без остатка на количество компьютеров
 * в системе.
 * В данном примере задачу запускаем на 4-х
 * компьютерах и на решетке 2x2.
 */
   #include<stdio.h>
   #include<stdlib.h>
   #include<mpi.h>
   #include<time.h>
   #include<sys/time.h>

/* NUM_DIMS - размер декартовой топологии.
 * "двумерная решетка" P0xP1 */

   #define NUM_DIMS 2
   #define P0 2
   #define P1 2

/* Задаем размеры матриц A = MxN, B = NxK и C =
MxK (Эти размеры значимы в ветви 0) */

   #define M 8
   #define N 8
   #define K 8
   #define A(i,j) A[N*i+j]
   #define B(i,j) B[K*i+j]
   #define C(i,j) C[K*i+j]

/*   Подпрограмма,   осуществляющая   перемножение
матриц */

PMATMAT_2(n, A, B, C, p, comm)
```

```
/* Аргументы A, B, C, n, p значимы в данном
случае только в ветви 0 */
    int *n;         /* Размеры исходных матриц */
/* Исходные матрицы:A[n[0]][n[1]], B[n[1]][n[2]],
  * C[n[0]][n[2]]; */
double *A, *B, *C;
/* Данные */
    int *p;
/* размеров решетки компьютеров. p[0]
 * соответствует n[0],
 * p[1] соответствует n[2]
 * и произведение p[0]*p[1] будет эквивалентно
 * размеру группы comm
 * Коммуникатор для процессов, участвующих в
 * умножении матрицы на матрицу */
    MPI_Comm comm;

  {
/* Далее все описываемые переменные значимы во
всех ветвях, в том числе и ветви 0 */
/* Локальные подматрицы (полосы) */
    double *AA, *BB, *CC;
/* Размеры полос в A и B и подматриц CC в C */
    int   nn[2];
/* Декартовы координаты ветвей */
    int coords[2];
    int rank;      /* Порядковые номера ветвей */
/* Смещения и размер подматриц CC для сборки в
корневом процессе (ветви) */
    int  *countc, *dispc, *countb, *dispb;
/* Типы данных и массивы для создаваемых типов */
    MPI_Datatype typeb, typec, types[2];
    int  blen[2];
    int i, j, k;
    int periods[2], remains[2];
    int sizeofdouble, disp[2];
```

```
 /* Коммуникаторы для 2D решетки, для подрешеток
1D, и копии коммуникатора comm */
    MPI_Comm comm_2D, comm_1D[2], pcomm;
 /* Создаем новый коммуникатор */
    MPI_Comm_dup(comm, &pcomm);
 /* Нулевая ветвь передает всем ветвям массивы
n[] и p[] */
    MPI_Bcast(n, 3, MPI_INT, 0, pcomm);
    MPI_Bcast(p, 2, MPI_INT, 0, pcomm);
 /* Создаем 2D решетку компьютеров размером
p[0]*p[1] */
    periods[0] = 0;
    periods[1] = 0;
    MPI_Cart_create(pcomm, 2, p, periods, 0,
                                    &comm_2D);
 /* Находим ранги и декартовы координаты ветвей в
этой решетке */
    MPI_Comm_rank(comm_2D, &rank);
    MPI_Cart_coords(comm_2D, rank, 2, coords);

 /* Нахождение коммуникаторов для подрешеток 1D
для рассылки полос матриц A и B */
    for(i = 0; i < 2; i++)
      { for(j = 0; j < 2; j++)
          remains[j] = (i == j);
          MPI_Cart_sub(comm_2D, remains,
                              &comm_1D[i]);
      }
 /* Во всех ветвях задаем подматрицы (полосы) */
 /* Здесь предполагается, что деление без остатка
  */
    nn[0] = n[0]/p[0];
    nn[1] = n[2]/p[1];

  #define AA(i,j) AA[n[1]*i+j]
  #define BB(i,j) BB[nn[1]*i+j]
  #define CC(i,j) CC[nn[1]*i+j]
```

```
 AA=(double*)malloc(nn[0]*n[1]*sizeof(double));
 BB=(double*)malloc(n[1]*nn[1]*sizeof(double));
 CC=(double*)malloc(nn[0]*nn[1]*sizeof(double));

 /* Работа нулевой ветви */
    if(rank == 0)
      {
     /*  Задание типа данных для вертикальной
полосы в  B. Этот тип создать необходимо, т.к. в
языке C массив в памяти располагается по строкам.
Для  массива  A  такой  тип  создавать  нет
необходимости, т.к. там передаются горизонтальные
полосы, а они в памяти расположены непрерывно.
*/
 MPI_Type_vector(n[1], nn[1], n[2], MPI_DOUBLE,
                                       &types[0]);
 /* и корректируем диапазон размера полосы */
 MPI_Type_extent(MPI_DOUBLE, &sizeofdouble);
        blen[0]  = 1;
        blen[1]  = 1;
        disp[0]  = 0;
        disp[1]  = sizeofdouble * nn[1];
        types[1] = MPI_UB;
        MPI_Type_struct(2, blen, disp, types,
                                        &typeb);
        MPI_Type_commit(&typeb);

     /*  Вычисление  размера  подматрицы  BB  и
смещений  каждой  подматрицы  в  матрице  B.
Подматрицы BB упорядочены в B в соответствии с
порядком номеров компьютеров в решетке, т.к.
массивы расположены в памяти по строкам, то
подматрицы BB в памяти (в B) должны располагаться
в следующей последовательности:  BB0, BB1,.... */
   dispb =  (int *)malloc(p[1] * sizeof(int));
   countb = (int *)malloc(p[1] * sizeof(int));
        for(j = 0; j < p[1]; j++)
          { dispb[j] = j;
            countb[j] = 1;
          }
```

```
/* Задание типа данных для подматрицы CC в C */
MPI_Type_vector(nn[0], nn[1], n[2], MPI_DOUBLE,
                                    &types[0]);

/* и корректируем размер диапазона */
MPI_Type_struct(2, blen, disp, types, &typec);
       MPI_Type_commit(&typec);

/* Вычисление размера подматрицы CC и смещений
каждой подматрицы в матрице C. Подматрицы CC
упорядочены в C в соответствии с порядком номеров
компьютеров в решетке, т.к. массивы расположены в
памяти по строкам, то подматрицы CC в памяти (в
C) должны располагаться в следующей
последовательности: CC0, CC1, CC2, CC3, CC4,
CC5, CC6, CC7. */

dispc=(int *)malloc(p[0]*p[1]*sizeof(int));
countc=(int *)malloc(p[0]*p[1]*sizeof(int));

       for(i = 0; i < p[0]; i++)
         { for(j = 0; j < p[1]; j++)
            { dispc[i*p[1]+j]=(i*p[1]*nn[0]+j);
              countc[i*p[1]+j] = 1;
             }
         }
     }             /* Нулевая ветвь завершает
подготовительную работу */

/* Вычисления (этапы указаны на рис. 8.5) */
/*   1.  Нулевая ветвь передает (scatter)
горизонтальные полосы матрицы A по x координате
*/
    if(coords[1] == 0)
      {
MPI_Scatter(A,nn[0]*n[1],MPI_DOUBLE,AA,nn[0]*n[1]
                      ,MPI_DOUBLE,0,comm_1D[0]);
     }

/*   2.  Нулевая ветвь передает (scatter)
горизонтальные полосы матрицы B по y координате
*/
```

```
    if(coords[0] == 0)
      { MPI_Scatterv(B,countb,dispb, typeb, BB,
           n[1]*nn[1], MPI_DOUBLE, 0,comm_1D[1]);
      }

/* 3. Передача подматриц  AA в измерении y */
MPI_Bcast(AA, nn[0]*n[1], MPI_DOUBLE, 0,
                                    comm_1D[1]);

/* 4. Передача подматриц  BB в измерении x */
    MPI_Bcast(BB, n[1]*nn[1], MPI_DOUBLE, 0,
                                     comm_1D[0]);

 /* 5. Вычисление подматриц CC в каждой ветви */

    for(i = 0; i < nn[0]; i++)
      { for(j = 0; j < nn[1]; j++)
          { CC(i,j) = 0.0;
            for(k = 0; k < n[1]; k++)
             { CC(i,j)=CC(i,j)+AA(i,k)*BB(k,j);
              }
          }
      }

 /* 6. Сбор всех подматриц CC в ветви 0 */
    MPI_Gatherv(CC, nn[0]*nn[1], MPI_DOUBLE, C,
              countc, dispc, typec, 0, comm_2D);

 /*   Освобождение   памяти   всеми   ветвями   и
завершение подпрограммы */
    free(AA);
    free(BB);
    free(CC);
    MPI_Comm_free(&pcomm);
    MPI_Comm_free(&comm_2D);
    for(i = 0; i < 2; i++)
      { MPI_Comm_free(&comm_1D[i]);
      }
    if(rank == 0)
      { free(countc);
        free(dispc);
```

```
        MPI_Type_free(&typeb);
        MPI_Type_free(&typec);
        MPI_Type_free(&types[0]);
      }
    return 0;
  }

 /* Главная программа */

int main(int argc, char **argv)
  {
    int        size, MyP, n[3], p[2], i, j, k;
    int        dims[NUM_DIMS], periods[NUM_DIMS];
    double     *A, *B, *C;
    int        reorder = 0;
    struct timeval tv1, tv2;
    int dt1;
    MPI_Comm   comm;

/* Инициализация библиотеки MPI */
    MPI_Init(&argc, &argv);

/*  Каждая ветвь узнает количество задач в
стартовавшем приложении */
    MPI_Comm_size(MPI_COMM_WORLD, &size);

/* и свой собственный номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP);

/*  Обнуляем массив dims и заполняем массив
periods для топологии "двумерная решетка" */
    for(i = 0; i < NUM_DIMS; i++)
     { dims[i] = 0;
       periods[i] = 0;
     }

 /*  Заполняем массив dims, где указываются
размеры двумерной решетки */
    MPI_Dims_create(size, NUM_DIMS, dims);

 /*  Создаем топологию "двумерная решетка" с
communicator(ом) comm */
    MPI_Cart_create(MPI_COMM_WORLD,NUM_DIMS,dims,
```

```
                        periods, reorder, &comm);
 /* В первой ветви выделяем в памяти место для
исходных матриц */
    if(MyP == 0)
      {
     /* Задаем размеры матриц и размеры двумерной
решетки компьютеров */
        n[0] = M;
        n[1] = N;
        n[2] = K;
        p[0] = P0;
        p[1] = P1;

A=(double *)malloc(M*N*sizeof(double));
B=(double *)malloc(N*K*sizeof(double));
C=(double *)malloc(M*K*sizeof(double));

/* Генерируем в первой ветви исходные матрицы A и
B, матрицу C обнуляем */
        for(i = 0; i < M; i++)
          for(j = 0; j < N; j++)
            A(i,j) = i+1;
        for(j = 0; j < N; j++)
          for(k = 0; k < K; k++)
            B(j,k) = 21+j;
        for(i = 0; i < M; i++)
          for(k = 0; k < K; k++)
            C(i,k) = 0.0;
    }  /* Подготовка матриц ветвью 0 завершина */

/*  Засекаем начало умножения матриц во всех
ветвях */
    gettimeofday(&tv1, (struct timezone*)0);

 / Все ветви вызывают функцию перемножения матриц
*/
    PMATMAT_2(n, A, B, C, p, comm);

 /* Умножение завершено. Каждая ветвь умножила
свою полосу строк матрицы A на полосу столбцов
матрицы B. Результат находится в нулевой ветви.
```

```
  * Засекаем время и результат печатаем */
    gettimeofday(&tv2, (struct timezone*)0);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
                      tv2.tv_usec - tv1.tv_usec;
    printf("MyP = %d Time = %d\n", MyP, dt1);
 /* Для контроля 0-я ветвь печатает результат */
    if(MyP == 0)
      { for(i = 0; i < M; i++)
          { for(j = 0; j < K; j++)
              printf(" %3.1f",C(i,j));
            printf("\n");
          }
      }
 /*  Все ветви завершают системные процессы,
связанные с топологией comm и завершают
выполнение программы */

    if(MyP == 0)
      { free(A);
        free(B);
        free(C);
      }
    MPI_Comm_free(&comm);
    MPI_Finalize();
    return(0);
  }
```

8.2.2.Параллельные алгоритмы решения систем линейных алгебраических уравнений методом Гаусса. Требуется найти решение системы линейных алгебраических уравнений:

$$Ax = f$$

Метод Гаусса основан на последовательном исключении неизвестных.

Здесь рассматриваются два алгоритма решения СЛАУ методом Гаусса. Они связаны с разными способами представления *данных* (матрицы коэффициентов и правых

частей) в распределенной памяти мультикомпьютера. Хотя данные распределены в памяти мультикомпьютера в каждом алгоритме по разному, но оба они реализуются на одной и той же топологии связи компьютеров - "полный граф". Топология "полный граф" обладает одной нехорошей особенностью, а именно, с ростом количества компьютеров в системе увеличивается и время выполнения коллективной операции обмена данными.

Поэтому при решении этой задачи нужно выбрать оптимальное соотношение между объемом данных и размером вычислительной системы. И это соотношение зависит от скорости обменов между компьютерами.

8.2.2.1 Первый алгоритм решения СЛАУ методом Гаусса. В алгоритме, представленном здесь, исходная матрица коэффициентов A и вектор правых частей F разрезаны горизонтальными полосами, как показано на рис. 8.6. Каждая полоса загружается в соответствующий компьютер: нулевая полоса – в нулевой компьютер, первая полоса – в первый компьютер, и т. д., последняя полоса – в p_1 компьютер.

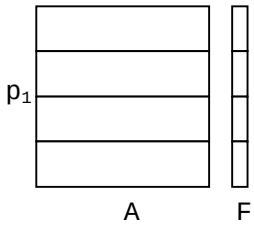


Рис. 8.6. Разрезание *данных* для параллельного алгоритма 1 решения СЛАУ методом Гаусса

При прямом ходе матрица приводится к треугольному виду последовательно по компьютерам. Вначале к треугольному виду приводятся строки в нулевом компьютере, при этом нулевой компьютер последовательно, строка за строкой, передает свои

сроки остальным компьютерам, начиная с первого. Затем к треугольному виду приводятся строки в первом компьютере, передавая свои строки остальным компьютерам, начиная со второго, т.е. компьютерам с большими номерами, и т. д. Процесс деления строк на коэффициенты при x_i не требует информации от других компьютеров.

После прямого хода полосы матрицы A в каждом узле будут иметь вид (рис. 8.7)

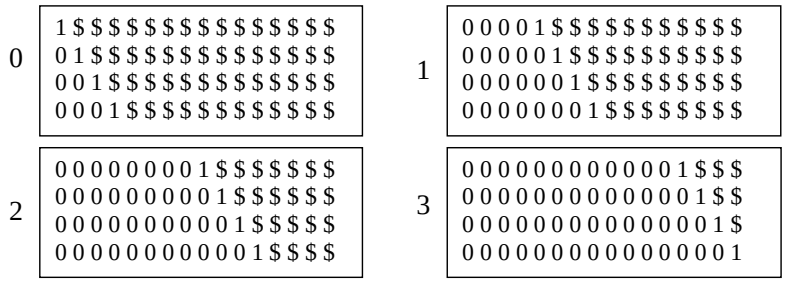


Рис. 8.7. Вид полос после прямого хода в алгоритме 1 решения СЛАУ методом Гаусса

Пример приведен для четырех узлов; $ – вещественные числа.

Аналогично, последовательно по компьютерам, начиная с последнего по номеру компьютера, осуществляется обратный ход.

Особенностью этого алгоритма является то, что как при прямом, так и при обратном ходе, компьютеры, завершившие свою часть работы, переходят в состояние ожидания, пока не завершат эту работу другие компьютеры. Таким образом, вычислительная нагрузка распределяется по компьютерам неравномерно, не смотря на то, что *данные* изначально распределяются по компьютерам приблизительно одинаково. Простои компьютеров значительно уменьшаются при

распределении матрицы циклическими горизонтальными полосами. Этот метод представлен в п. 2.2.2.

Здесь, в примере, каждая ветвь генерирует свои части матрицы.

```
/* Первый алгоритм
 *  Решение СЛАУ методом Гаусса. Распределение данных - горизонтальными полосами.
 * (Запуск задачи на 8-ми компьютерах).
 */
   #include<stdio.h>
   #include<mpi.h>
   #include<sys/time.h>

/*  Каждая ветвь задает размеры своих полос матрицы MA и вектора правой части.
 * (Предполагаем, что размеры данных делятся без остатка на количество компьютеров.) */

   #define M 400
   #define N 50

   #define tegD 1
   #define EL(x) (sizeof(x) / sizeof(x[0]))

/* Описываем массивы для полос исходной матрицы - MA и вектор V для приема данных. Для простоты, вектор правой части уравнений присоединяем дополнительным столбцом к матрице коэффициентов. В этом дополнительном столбце и получим результат. */

double MA[N][M+1], V[M+1], MAD, R;

int main(int args, char **argv)
  { int size, MyP, i, j, v, k, d, p;
    int       *index, *edges;
    MPI_Comm   comm_gr;
    MPI_Status status;
    struct timeval tv1, tv2;
    int dt1;
    int reord = 1;
```

```
/* Инициализация библиотеки */
    MPI_Init(&args, &argv);
/* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* и свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP);
/* Выделяем память под массивы для описания
вершин и ребер в топологии полный граф */
index = (int *)malloc(size * sizeof(int));
edges=(int *)malloc(size*(size-1)*sizeof(int));
/* Заполняем массивы для описания вершин и ребер
для топологии полный граф и задаем топологию
"полный граф". */
    for(i = 0; i < size; i++)
      { index[i] = (size - 1)*(i + 1);
        v = 0;
        for(j = 0; j < size; j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j;
          }
      }
   MPI_Graph_create(MPI_COMM_WORLD, size, index,
                         edges, reord, &comm_gr);
/* Каждая ветвь генерирует свою полосу матрицы A
и свой отрезок вектора правой части, который
присоединяется дополнительным столбцом к A.
Нулевая ветвь генерирует нулевую полосу, первая
ветвь - первую полосу и т.д. (По диагонали
исходной матрицы - числа = 2, остальные числа =
1). */
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if((N*MyP+i) == j)
              MA[i][j] = 2.0;
            else
```

```
            MA[i][j] = 1.0;
          }
        MA[i][M] = 1.0*(M)+1.0;
      }
```

/* Каждая ветвь засекает начало вычислений и производит вычисления */

```
    gettimeofday(&tv1, (struct timezone*)0);
```

/* Прямой ход */

/* Цикл p - цикл по компьютерам. Все ветви, начиная с нулевой, последовательно приводят к диагональному виду свои строки. Ветвь, приводящая свои строки к диагональному виду, назовем *активной*, строка, с которой производятся вычисления, так же назовем *активной*. */

```
    for(p = 0; p < size; p++)
      {
```

/* Цикл k - цикл по строкам. (Все веиви "крутят" этот цикл). */

```
     for(k = 0; k < N; k++)
       { if(MyP == p)
          {
```

/* Активная ветвь с номером MyP == p приводит свои строки к диагональному виду. Активная строка k передается ветвям, с номером большим чем MyP */

```
            MAD = 1.0/MA[k][N*p+k];
            for(j = M; j >= N*p+k; j--)
             MA[k][j] = MA[k][j] * MAD;
            for(d = p+1; d < size; d++)
              MPI_Send(&MA[k][0], M+1,
            MPI_DOUBLE, d, tegD, comm_gr);
             for(i = k+1; i < N; i++)
               { for(j = M; j >= N*p+k; j--)
                   MA[i][j] = MA[i][j]-
                     MA[i][N*p+k]*MA[k][j];
                }
          }
```

```
/* Работа принимающих ветвей с номерами MyP > p
*/

          else if(MyP > p)
            {  MPI_Recv(V, EL(V), MPI_DOUBLE, p,
                        tegD, comm_gr, &status);
                for(i = 0; i < N; i++)
                  { for(j = M; j >= N*p+k; j--)
                      MA[i][j] = MA[i][j]-
                          MA[i][N*p+k]*V[j];
                  }
             }
         }          /* for k */
      }             /* for p */

/* Обратный ход */

/* Циклы по p и k аноалогичны, как и при прямом
ходе. */

    for(p = size-1; p >= 0; p--)
      { for(k = N-1; k >= 0; k--)
          {
      /* Работа активной ветви */

            if(MyP == p)
              { for(d = p-1; d >= 0; d--)
                 MPI_Send(&MA[k][M], 1,
                  MPI_DOUBLE, d, tegD, comm_gr);
                for(i = k-1; i >= 0; i--)
                MA[i][M]-=MA[k][M]*MA[i][N*p+k];
              }

      /* Работа ветвей с номерами MyP < p */

           else
             { if(MyP < p)
                 { MPI_Recv(&R, 1, MPI_DOUBLE, p,
                         tegD, comm_gr, &status);
                    for(i = N-1; i >= 0; i--)
                      MA[i][M] -= R*MA[i][N*p+k];
                 }
              }
          }             /* for k */
```

```
      }                    /* for p */
/* Все ветви засекают время и печатают */
    gettimeofday(&tv2, (struct timezone*)0);
dt1 = (tv2.tv_sec - tv1.tv_sec)*1000000 +
                     tv2.tv_usec - tv1.tv_usec;
    printf("MyP = %d Time = %d\n", MyP, dt1);
/* Все ветви печатают, для контроля, свои первые
четыре значения корня */
printf("MyP = %d %f %f %f %f\n", MyP, MA[0][M],
                     MA[1][M],MA[2][M],MA[3][M]);
/* Все ветви завершают выполнение */
    MPI_Finalize();
    return(0);
  }
```

8.2.2.2 Второй алгоритм решения СЛАУ методом Гаусса. В алгоритме, представленном здесь, исходная матрица коэффициентов распределяется по компьютерам циклическими горизонтальными полосами с шириной полосы в одну строку, как показано ниже на рис. 8.8.

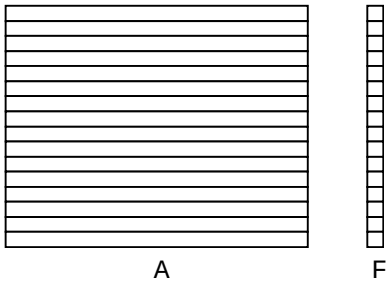


Рис. 8.8. Разрезание *данных* для параллельного алгоритма 2 решения СЛАУ методом Гаусса.

Первая строка матрицы помещается в компьютер 0, вторая строка – в компьютер 1, и т. д., (p_1-1)-я строка в компьютер p_1 (где p_1 количество компьютеров в системе). Затем, p1–я строка,

снова помещается в компьютер 0, $p1+1$-я строка – в компьютер 1, и т. д.

При таком распределении данных, соответствующим этому распределению должен быть и алгоритм. Строку, которая вычитается из всех остальных строк (после предварительного деления на нужные коэффициенты), назовем текущей строкой. Алгоритм прямого хода заключается в следующем. Сначала текущей строкой является строка с индексом 0 в компьютере 0, затем строка с индексом 0 в компьютере 1 (здесь не нужно путать общую нумерацию строк во всей матрице и индексацию строк в каждом компьютере; в каждом компьютере индексация строк в массиве начинается с нуля) и т. д., и наконец, строка с индексом 0 в последнем по номеру компьютере. После чего цикл по компьютерам повторяется и текущей строкой становится строка с индексом 1 в компьютере 0, затем строка с индексом 1 в компьютере 1 и т. д. После прямого хода полосы матрицы в каждом компьютере будут иметь вид, показанный на рис. 8.9. Пример приведен для четырех узлов; $ – вещественные числа.
Аналогично, последовательно по узлам, начиная с последнего по номеру компьютера, осуществляется обратный ход.

Особенностью этого алгоритма является то, что как при прямом, так и при обратном ходе компьютеры являются более равномерно загруженными, чем в первом методе. Значит, и вычислительная нагрузка распределяется по компьютерам более равномерно, чем в первом методе. Например, нулевой компьютер, завершив обработку своих строк при прямом ходе, ожидает, пока другие компьютеры обработают только по одной, оставшейся у них не обработанной строке, а не полностью обработают полосы, как в первом алгоритме.

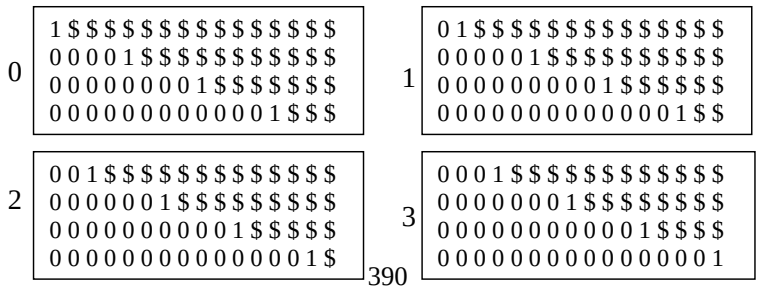


Рис. 8.9. Вид полос после прямого хода в алгоритме 2 решения СЛАУ методом Гаусса.

Сравним второй алгоритм с первым. При более равномерной загрузке компьютеров при вычислении одного алгоритма по сравнению с другим алгоритмом следует предположить и большую эффективность алгоритма с более равномерной загрузкой компьютеров. Загрузка компьютеров во втором алгоритме является более равномерной. Но большая его эффективность, по сравнению с первым, может проявиться только на исходных матрицах большого размера (например, начиная с исходных матриц 400 х 400 и более, но это зависит от конкретной системы). Это обстоятельство связано с тем, что в первом методе в процессе вычислений активных компьютеров становится все меньше, а значит, и уменьшается количество пересылок своих строк другим компьютерам. С уменьшением числа активных компьютеров будет уменьшаться и общее время, затрачиваемое на пересылку строк в активные компьютеры. И это частично компенсирует неравномерность вычислительной загрузки компьютеров. Во втором методе компьютеры активны в течение всего времени вычислений и пересылка строк осуществляется всегда во все компьютеры. Затраты на пересылку одной строки из разных компьютеров, в этом случае, будут всегда максимальны.

```
/* Второй алгоритм
 *  Решение СЛАУ методом Гаусса. Распределение
данных - циклическими горизонтальными полосами.
 * (Запуск задачи на 8-ми компьютерах).
 */
   #include<stdio.h>
   #include<mpi.h>
   #include<sys/time.h>

/*  Каждая ветвь задает размеры своих полос
матрицы MA и вектора правой части.
```

```
 * (Предполагаем, что размеры данных делятся без
остатка на количество компьютеров.) */

   #define M 400
   #define N 50

   #define tegD 1

/*  Описываем массив для циклических полос
исходной матрицы - MA и вектор V для приема
данных. Для простоты, вектор правой части
уравнений присоединяем дополнительным столбцом к
матрице коэффициентов. В этом дополнительном
столбце и получим результат. */

double MA[N][M+1], V[M+1], MAD, R;

int main(int args, char **argv)
  { int        size, MyP, i, j, v, k, k1, p;
    int       *index, *edges;
    MPI_Comm   comm_gr;
    struct timeval tv1, tv2;
    int dt1;
    int reord = 1;

/* Инициализация библиотеки */

    MPI_Init(&args, &argv);

/* Каждая ветвь узнает размер системы */

    MPI_Comm_size(MPI_COMM_WORLD, &size);

/* и свой номер (ранг) */

    MPI_Comm_rank(MPI_COMM_WORLD, &MyP);

/*  Выделяем память под массивы для описания
вершин и ребер в топологии полный граф */

  index = (int *)malloc(size * sizeof(int));
  edges=(int*)malloc(size*(size-1)*sizeof(int));

 /* Заполняем массивы для описания вершин и ребер
для топологии полный граф и задаем топологию
"полный граф". */
```

```
    for(i = 0; i < size; i++)
      { index[i] = (size - 1)*(i + 1);
        v = 0;
        for(j = 0; j < size; j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j;
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index,
                          edges, reord, &comm_gr);
```

/* Каждая ветвь генерирует свои циклические полосы матрицы A и свой отрезок вектора правой части, который присоединяется дополнительным столбцом к A.
  * Нулевая ветвь генерирует следующие строки исходной матрицы: 0, size, 2*size, 3*size, и т.д. Первая ветвь - строки: 1, 1+size, 1+2*size, 1+3*size и т.д. Вторая ветвь - строки: 2, 2+size, 2+2*size, 2+3*size и т.д. (По диагонали исходной матрицы - числа = 2, остальные числа = 1). */

```
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if((MyP+size*i) == j)
              MA[i][j] = 2.0;
            else
              MA[i][j] = 1.0;
          }
        MA[i][M] = 1.0*(M)+1.0;
      }
```

/* Каждая ветвь засекает начало вычислений и производит вычисления */

```
    gettimeofday(&tv1, (struct timezone*)0);
```

/* Прямой ход */

/* Цикл k - цикл по строкам. Все ветви, начиная с нулевой, последовательно приводят к диагональному виду свои строки. Ветвь, приводящая свои строки к диагональному виду, назовем

активной, строка, с которой производятся вычисления, так же назовем *активной*. */

```
    for(k = 0; k < N; k++)
      {
/* Цикл p - цикл по компьютерам. (Все веиви
"крутят" этот цикл). */
        for(p = 0; p < size; p++)
          { if(MyP == p)
              {
/* Активная ветвь с номером MyP == p приводит
свою строку с номером k к диагональному виду.
 * Активная строка - k передается всем ветвям. */
               MAD = 1.0/MA[k][size*k+p];
               for(j = M; j >= size*k+p; j--)
                 MA[k][j] = MA[k][j] * MAD;
               for(j = 0; j <= M; j++)
                 V[j] = MA[k][j];
           MPI_Bcast(V,M+1,MPI_DOUBLE,p,comm_gr);
               for(i = k+1; i < N; i++)
                 { for(j = M; j >= size*k+p; j--)
                     MA[i][j]=MA[i][j]-
                       MA[i][size*k+p]*MA[k][j];
                 }
              }
/* Работа принимающих ветвей с номерами MyP < p
*/
         else if(MyP < p)
           { MPI_Bcast(V,M+1,MPI_DOUBLE,p,
                                       comm_gr);
             for(i = k+1; i < N; i++)
              { for(j = M; j >= size*k+p; j--)
                  MA[i][j]=MA[i][j]-
                            MA[i][size*k+p]*V[j];
               }
             }
/* Работа принимающих ветвей с номерами MyP > p
*/
```

```
            else if(MyP > p)
              { MPI_Bcast(V, M+1, MPI_DOUBLE, p,
                                          comm_gr);
                 for(i = k; i < N; i++)
                   { for(j=M;j >= size*k+p;j--)
                      MA[i][j] = MA[i][j] –
                         MA[i][ size*k+p]*V[j];
                    }
                 }
          }          /*for p */
      }              /*for k */
```

/* Обратный ход */

/* Циклы по k и p аноалогичны, как и при прямом ходе. */

```
     for(k1 = N-2, k = N-1; k >= 0; k--,k1--)
       { for(p = size-1; p >= 0; p--)
           { if(MyP == p)
               {
              /* Работа активной ветви */
                R = MA[k][M];
            MPI_Bcast(&R,1,MPI_DOUBLE,p,comm_gr);
                for(i = k-1; i >= 0; i--)
                  MA[i][M]-= MA[k][M]*
                             MA[i][size*k+p];
               }
           /* Работа ветвей с номерами MyP < p */
              else if(MyP < p)
                {
             MPI_Bcast(&R,1,MPI_DOUBLE,p,comm_gr);
                 for(i = k; i >= 0; i--)
                   MA[i][M] -= R*MA[i][ size*k+p];
               }
           /* Работа ветвей с номерами MyP > p */
              else if(MyP > p)
               {
             MPI_Bcast(&R,1,MPI_DOUBLE,p,comm_gr);
```

```
                for(i = k1; i >= 0; i--)
                  MA[i][M] -= R*MA[i][size*k+p];
              }
            }          /* for p */
        }              /* for k */

/* Все ветви засекают время и печатают */
    gettimeofday(&tv2, (struct timezone*)0);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
                      tv2.tv_usec - tv1.tv_usec;
    printf("MyP = %d Time = %d\n", MyP, dt1);

/* Все ветви печатают, для контроля, свои первые
четыре значения корня */
printf("MyP = %d %f %f %f %f\n", MyP, MA[0][M],
                    MA[1][M],MA[2][M],MA[3][M]);

 /* Все ветви завершают выполнение */
    MPI_Finalize();
    return(0);
  }
```

8.2.3.Параллельные алгоритмы решения СЛАУ итерационными методами. Здесь рассматриваются параллельные алгоритмы решения СЛАУ методом простой итерации и методом сопряженных градиентов. Данными в этих алгоритмах являются матрицы и вектора. Распараллеливаются эти алгоритмы методом декомпозиции данных.

8.2.3.1 Параллельный алгоритм решения СЛАУ методом простой итерации. Дана система линейных алгебраических уравнений:

$$Ax = f$$

Корни этой системы методом простой итерации вычисляются по следующей формуле:

$$x_i^{k+1} = x_i^k - \tau * (\sum_{j=0}^{N} a_{ij} x_j^k - f_i) .... i = 0,..., N$$

Здесь верхний индекс обозначает номер итерационного шага, i - номер корня и правой части. τ – итерационный шаг.
Завершаются вычисления по условию:

$$\frac{\left\| \sum_{j=0}^{N} a_{ij} x_{ij}^k - f_i \right\|}{\|f\|} < \varepsilon \qquad (1)$$

Для решения этой задачи на параллельной системе исходную матрицу коэффициентов A разрезаем на p_1 горизонтальных полосы по строкам, где p_1 – количество компьютеров в системе. Аналогично, горизонтальными полосами разрезаются вектор f (правая часть) и вектора x_0 (начальное приближение), x_k (текущее приближение) и x_{k+1} (следующее приближение). Полосы последовательно распределяются по соответствующим компьютерам системы, как и в описанном выше, первом алгоритме умножения матрицы на матрицу.

Поскольку матрица коэффициентов и правые части разрезаны, то в каждом компьютере системы вычисляется "свое" подмножество корней и частичная сумма невязки (числитель условия 1). Поэтому после нахождения приближенных значений корней на очередном шаге итерации в каждом компьютере, перед тем как проверяется выполнение условия завершения вычислений, предварительно частичные суммы невязки суммируются по всем компьютерам. После суммирования частичных сумм невязки по компьютерам выражение условия выхода (1) будет одним и тем же на всех компьютерах на каждом шаге итерации. Поэтому все ветви параллельной программы будут продолжать или завершать вычисления одинаково.

```
/*
```

```
 *   Решение СЛАУ методом простой итерации.
Распределение данных - горизонтальными полосами.
(Запуск примера на 8-ми компьютерах). */

#include<stdio.h>
#include<mpi.h>
#include<sys/time.h>

/* Каждая ветвь задает размеры своих полос матрицы
MA и вектора правой части. (Предполагаем, что
размеры данных делятся без остатка на количество
компьютеров.) */

#define M 64
#define N 8

/* Задаем необходимую точность приближения корней
*/

#define E 0.00001

/* Задаем шаг итерации */

#define T 0.01

/*Описываем массивы для полос исходной матрицы -MA,
вектора правой части - F, значения приближений на
предыдущей итерации - Y и текущей - Y1, результата
умножения матрицы коэффициентов на вектор - S, и
всего вектора значения приближений на предыдущей
итерации - V. */

static double MA[N][M], F[N], Y[N], Y1[N], S[N] ;
static double V[M], Fm, Fmp, Xm, Xmp;

int main(int argc, char **argv)
  { int i, j, z, rank, size, v, it;
    int *index, *edges;
    MPI_Comm  comm_gr;
    struct timeval tv1, tv2;
    int dt1, reord = 1;

  /* Инициализация библиотеки */
```

```
    MPI_Init(&argc, &argv);
  /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
  /* и свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  /* Выделяем память под массивы для описания вершин и
  ребер в топологии полный граф */
   index = (int *)malloc(size * sizeof(int));
   edges=(int *)malloc(size*(size-1)*sizeof(int));
  /* Заполняем массивы для описания вершин и ребер
  для топологии полный граф и задаем топологию
  "полный граф". */
    for(i = 0; i < size; i++)
      { index[i] = (size - 1)*(i + 1);
        v = 0;
        for(j = 0; j < size; j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j;
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index,
edges, reord, &comm_gr);
  /* Каждая ветвь генерирует свои полосы матрицы A
  и свой отрезок вектора правой части.
   * (По диагонали исходной матрицы - числа = 2,
остальные числа = 1). */
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if((N*rank + i) == j)
              MA[i][j] = 2.0;
            else
              MA[i][j] = 1.0;
          }
        F[i] = M + 1;
      }
/*  Каждая ветвь задает начальное приближение
корней. */
```

```
    for(Fm = 0,i = 0; i < N; i++)
      { Y1[i] = 0.8;
        Fm += F[i] * F[i];
      }
/* Находим ||f||. Суммируем Fm по всем компьютерам
и запоминаем в каждом. Каждая параллельная ветвь
будет иметь ||f||.  */
MPI_Allreduce(&Fm,&Fmp,1,MPI_DOUBLE,MPI_SUM,
                                        comm_gr);

    gettimeofday(&tv1,NULL);

    it = 0;
/* Начало вычислений. Главный цикл. */
    do
      { for(i = 0; i < N; i++)
         Y[i] = Y1[i];
/*  В каждой ветви формируем весь вектор предыдущей
итерации и умножаем матрицу коэффициентов на этот вектор
*/
        MPI_Allgather(Y, N, MPI_DOUBLE, V, N,
                        MPI_DOUBLE, comm_gr);
        for(Xm = 0, i = 0; i < N; i++)
          { for(S[i] = 0, j = 0; j < M; j++)
             S[i] += MA[i][j] * V[j];
          Y1[i] = Y[i] - T*(S[i] - F[i]);
            Xm += (S[i] - F[i]) * (S[i] - F[i]);
          }
        it++;
      MPI_Allreduce(&Xm, &Xmp, 1, MPI_DOUBLE,
                              MPI_SUM, comm_gr);
      }
    while(Xmp/Fmp > E*E);
/* конец основного цикла */
/* Все ветви засекают время и его значение выводят
на монитор */
    gettimeofday(&tv2, NULL);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
```

```
                        tv2.tv_usec - tv1.tv_usec;
    printf(" rank = %d Time = %d\n", rank, dt1);
/* Все ветви печатают, для контроля, свои первые
четыре значения корня */
    printf(" rank = %d Y0=%f Y1=%f Y2=%f Y3=%f\n",
                         rank,Y[0],Y[1],Y[2],Y[3]);
/* Все ветви завершают вычисления */
    MPI_Finalize();
    return(0);
  }
```

8.2.3.2 Параллельный алгоритм решения СЛАУ методом сопряженных градиентов. Дана система линейных алгебраических уравнений:

$$Ax = f$$

Этот метод предназначен для симметричных матриц:

$$a_{ij} = a_{ji}.$$

Выбирается начальное приближение вектора решений: x_0.

$x_k = x_{k-1} + \alpha_k z_{k-1}$ – вычисление вектора решений на к-й текущей итерации;

$r_0 = f - Ax_0$ – начальное приближение вектора невязки;

$r_k = r_{k-1} - \alpha_k A z_{k-1}$ – вычисление вектора невязки на *к*-й текущей итерации;

$\alpha_k = (r_{k-1}, r_{k-1})/(Az_{k-1}, z_{k-1})$ – коэффициент (здесь *к* – это номер итерационного шага);

$z_0 = r_0$ – начальное приближение вектора спуска;

$z_k = r_k + \beta_k z_{k-1}$ – вычисление вектора спуска на *к*-й текущей итерации (сопряженное направление).

$\beta_k = (r_k, r_k)/(r_{k-1}, r_{k-1})$ – коэффициент;

Условие выхода из итерационного процесса:

$$\frac{\|r_k\|}{\|f\|} < \varepsilon$$

Для решения этой задачи на параллельной системе исходную матрицу коэффициентов A разрезаем на p_1 горизонтальных полосы по строкам, где p_1 – количество компьютеров в системе. Аналогично, горизонтальными полосами разрезаются вектор f (правая часть) и вектора x_k, r_k, z_k. Полосы последовательно распределяются по соответствующим компьютерам системы. Завершение вычислений аналогично, как и в предыдущем алгоритме.

```
/*
 * Решение СЛАУ методом сопряженных градиентов. Распределение данных - горизонтальными полосами. (Запуск примера на 8-ми компьютерах). */
#include<stdio.h>
#include<mpi.h>
#include<time.h>
#include<sys/time.h>

/* Каждая ветвь задает размеры своих полос матрицы MA и вектора правой части. (Предполагаем, что размеры данных делятся без остатка на количество компьютеров.) */

#define M 64
#define N 8

/* Задаем необходимую точность приближения корней */
#define E 0.00001

/* Задаем массивы исходных и промежуточных данных */
static double A[N][M], F[N], Xk[N], Zk[N], Pk[M];
```

```
static double Rk[N],Rkp[M],Sr[N],alf,bet, mf, mfp;
static double Spr1, Spr1p, Spp, Sppr, Sppp, Spprp;

int main(int argc, char **argv)
  { int i, j, v, rank, size;
    int *index, *edges;
    MPI_Comm    comm_gr;
    struct timeval tv1, tv2;
    long int dt1;
    int reord = 1;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    index = (int *)malloc(size * sizeof(int));
    edges=(int*)malloc(size*(size-1)*sizeof(int));

    for(i = 0; i < size; i++)
      { index[i] = (size - 1)*(i + 1);
        v = 0;
        for(j = 0; j < size; j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j;
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index,
                          edges, reord, &comm_gr);
/* Генерация данных */
    for(mf=0,i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if((N*rank + i) == j)
              A[i][j] = 2.0;
            else
              A[i][j] = 1.0;
          }
        F[i] = M + 1;
      mf += F[i] * F[i];
```

```
      }
/* Находим ||f|| у всех ветвей. */
     MPI_Allreduce(&mf,&mfp,1,MPI_DOUBLE,MPI_SUM,
                                    MPI_COMM_WORLD);
/* Каждая параллельная ветвь задает начальное
приближение Xk, rk, Zk. */
    for(i = 0; i < M; i++)
      Rkp[i] = 0.2;

    for(i = 0; i < N; i++)
      Xk[i] = 0.2;

    for(i = 0; i < N; i++)
      { for(Sr[i]=0,j = 0; j < M; j++)
          Sr[i] += A[i][j] * Rkp[j];
      Rk[i] = F[i] - Sr[i];
      Zk[i] = Rk[i];
      }
    MPI_Allgather(Zk, N, MPI_DOUBLE, Rkp, N,
                         MPI_DOUBLE, comm_gr);
        for(i = 0; i < N; i++)
         { for(Pk[i]=0,j = 0; j < M; j++)
             Pk[i] += A[i][j] * Rkp[j];
         }

    gettimeofday(&tv1, NULL);
/* Начало вычислений. Главный цикл. */
    do
      {
        Spp = 0;
      Sppr = 0;
        for(i = 0; i < N; i++)
          { Spp += Pk[i] * Pk[i];
          Sppr += Pk[i] * Rk[i];
          }
        MPI_Allreduce(&Spp, &Sppp, 1, MPI_DOUBLE,
                                  MPI_SUM, comm_gr);
        MPI_Allreduce(&Sppr, &Spprp, 1, MPI_DOUBLE,
                                  MPI_SUM, comm_gr);
```

```
        alf = Spprp/Sppp;

        Spr1 = 0;
        for(i = 0; i < N; i++)
          { Xk[i] += alf*Zk[i];
            Rk[i] -= alf*Pk[i];
            Spr1 += Rk[i]*Rk[i];
          }
        MPI_Allreduce(&Spr1, &Spr1p, 1, MPI_DOUBLE,
                                  MPI_SUM, comm_gr);

        MPI_Allgather(Rk, N, MPI_DOUBLE, Rkp, N,
                              MPI_DOUBLE, comm_gr);

        for(Sppr=0,i = 0; i < N; i++)
         { for(Sr[i]=0, j = 0; j < M; j++)
             Sr[i] += A[i][j] * Rkp[j];
           Sppr += Pk[i] * Sr[i];
         }
        MPI_Allreduce(&Sppr, &Spprp, 1, MPI_DOUBLE,
                                   MPI_SUM, comm_gr);
      bet = Spprp/Sppp;

        for(i = 0; i < N; i++)
         { Zk[i] = Rk[i] + bet * Zk[i];
           Pk[i] = Sr[i] + bet * Pk[i];
         }
      }
    while(Spr1p/mfp > E*E);
/* конец основного цикла */

/* Все ветви засекают время и его значение выводят
на монитор */

    gettimeofday(&tv2, NULL);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
tv2.tv_usec - tv1.tv_usec;
    printf("rank = %d Time = %d\n", rank, dt1);

/* Все ветви печатают, для контроля, свои первые
восемь значений корней */
```

```
printf("rank=%d  %f %f %f %f %f %f %f %f\n",rank,
  Xk[0],Xk[1],Xk[2],Xk[3],Xk[4],Xk[5],Xk[6],Xk[7]);
/* Все ветви завершают вычисления */
    MPI_Finalize();
    return(0);
  }
```

8.3.Программирование на суперкомпьютерах с общей памятью и примеры параллельных программ в OpenMP

В этой части приводятся примеры параллельных алгоритмов решения задач на вычислительных системах с *общей памятью* с использованием системы параллельного программирования OpenMP. OpenMP в настоящее время является основной системой, ориентированным на параллельные вычисления над общей памятью. Приведенные здесь параллельные программы решения задач являются иллюстрационными, демонстрирующими способы распараллеливания задач подобного класса, и показывающими применение и возможности директив OpenMP. Рассматриваемые алгоритмы задач распараллеливаются, в основном, среднезернистыми методами.

OpenMP имеет явную (не автоматическую) модель программирования, предлагая программисту полное управление по распараллеливанию. В рамках данной технологии директивы параллелизма используются для выделения в программе *параллельных областей* (*parallel regions*), в которых последовательный исполняемый код может быть разделен на несколько раздельных командных *потоков* (*threads*). Далее эти потоки могут исполняться на разных процессорах вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков программного кода (см. рис. 3.1). Подобный принцип организации параллелизма получил наименование "*вилочного*" (*Fork-Join*) или *пульсирующего параллелизма*. Более полная информация по технологии OpenMP может быть получена, например, в информационных ресурсах сети Интернет [21].

Fork (ВЕТВЛЕНИЕ): главный поток создает группу параллельных потоков.

Код программы, который включен в параллельную конструкцию, будет выполняться параллельно в различных потоках.

Join (ОБЪЕДИНЕНИЕ): когда потоки в параллельной области завершаются, они синхронизируются и закрываются, оставляя только главный поток (рис. 8.10).

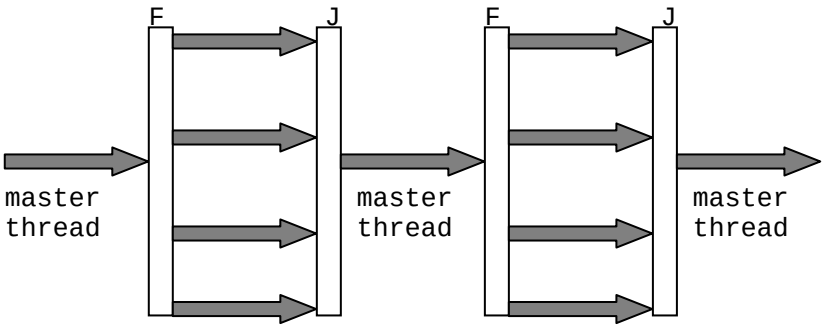


Рис. 8.10. Модель распараллеливания в OpenMP

Директивой, задающей параллельный блок, является директива:

```
#pragma omp parallel [clause clause ...]
   { . . .
   }
```

Основными директивами, которые распределяют работу по параллельным процессам, являются:

```
#pragma omp for [clause clause ...]
      { . . .
      }
```

и

```
#pragma omp sections [clause clause ...]
     { . . .
#pragma omp section
       { . . .
       }
     }
```

Директива `for` распараллеливает циклы. Витки циклов выполняются по одному и тому же алгоритму, т.е. директива `#pragma omp for` использует SPMD-модель вычислений. Но SPMD-модель не что иное, как *распараллеливание по данным*. Т.е. каждый параллельно исполняемый процесс обрабатывает некоторую часть назначенных ему данных аналогично, как и в MPI, но разница только в том, что здесь данные расположены в общей памяти.

Директива `#pragma omp sections` как правило применяется для параллельного выполнения некоторых участков программы с разными кодами, т.е. эта директива использует MPMD-модель вычислений.

8.3.1. Умножение матрицы на матрицу. Умножение матрицы на вектор и матрицы на матрицу являются, как было сказано, базовыми макрооперациями для многих задач линейной алгебры, например итерационных методов решения систем линейных уравнений и т. п. Поэтому приведенный алгоритм можно рассматривать как фрагмент в алгоритмах других задач. В отличие от программирования в MPI здесь рассматривается только одна схема распараллеливания алгоритма – это распараллеливание циклов (SPMD-модель вычислений). Такая схема определяется наличием общей памяти и независимостью выполнения циклов.

Заданы две исходные матрицы `A` и `B`. Вычисляется произведение `C = A x B`, где `A` - матрица $n_1 \times n_2$, и `B` - матрица $n_2 \times n_3$. Матрица результатов `C` имеет размер $n_1 \times n_3$. Исходные матрицы определены в общей памяти. Но распараллеливание циклов – это *распараллеливание по данным*. Т.е. каждый параллельно исполняемый процесс обрабатывает некоторую часть назначенных ему данных аналогично, как и в MPI, но разница только в том, что здесь данные расположены в общей памяти. Распараллеливается внешний цикл, переменная, которого указывает на строки матриц `A` и `C`. Поэтому матрицы `A` и `C` можно представить условно разрезанными, как показано на рис. 3.2, на `p` горизонтальных полос (`p` – количество

параллельных процессов). Матрица B не разрезана, т.к. столбцы этой матрицы указываются переменной внутреннего цикла, которые в данном случае не распараллеливается, т.е. все столбцы обрабатываются каждым параллельным процессом.

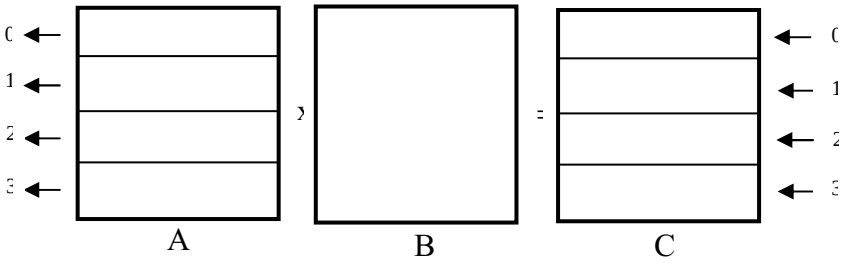


Рис. 8.11. Умножение матриц. Полосами указаны части исходных данных, обрабатываемых разными процессами (p=4).

Текст программы, реализующий алгоритм, приведен ниже.

```
/* Пример программы на языке C (с OpenMP)
произведения двух матриц. */
/* Здесь в примере исходные матрицы генерируются в
программе */

#include<stdio.h>
#include<sys/time.h>
#include<omp.h>

#define M 1000
#define N 1000

/* Задаем исходные матрицы. */

double A[N][M], B[N][M], C[N][M];

int main()
   { int i, j, v, nr;
     int size, rank;
     long int dt1;
```

```
     struct timeval tv1, tv2;

/* Генерация исходных матриц A и B, полосы C
обнуляются */
     for(i = 0; i < N; i++)
       { for(j = 0; j < M; j++)
           { A[i][j] = 3.0;
             B[j][i] = 2.0;
             C[i][j] = 0.0;
           }
       }

/* Заказываем количество параллельных процессов */
     omp_set_num_threads(4);

/* Задаем параллельный блок из (4-х) процессов.
Номер процесса (rank) должен быть обязательно
приватным во всех процессах*/
#pragma omp parallel private(rank,size,i)

 { /* Каждый процесс узнает количество процессов и
свой номер в стартовавшем приложении */
     size = omp_get_num_threads();
     rank = omp_get_thread_num();
     nr = M/size;
    gettimeofday(&tv1, NULL);

/* Задание работ. Распределяем витки внешнего цикла
по процессам и каждый процесс перемножает свои
полосы матрицы A */
#pragma omp for schedule(static,nr) private(j,v)\\
nowait
     for(i = 0; i < N; i++)
       { for(j = 0; j < N; j++)
         {  for(v = 0; v < M; v++)
                C[i][j] += A[i][v] * B[v][j];
           }
        }

/* Каждый процесс выводит время решения */

     gettimeofday(&tv2, NULL);
     dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
                       tv2.tv_usec - tv1.tv_usec;
```

```
    printf(" rank=%d  Time=%ld\n",rank,dt1);

  }  /* Конец параллельного блока */
}
```

8.3.2.Параллельный алгоритм решения СЛАУ методом Гаусса

Требуется найти решение системы линейных алгебраических уравнений:

$$Ax = f$$

Рассматриваемая здесь схема распараллеливания алгоритма, как и в предыдущей задаче, – это распараллеливание циклов. Это значит, что каждому процессу назначается для обработки полоса строк матрицы коэффициентов (см. рис. 8.12). Но при распараллеливании этого алгоритма в OpenMP есть особенности, существенно отличающие его от предыдущего алгоритма умножения матриц. При прямом ходе в данном алгоритме выбор строки (назовем ее текущей), которая в начале делится на диагональный коэффициент, а затем вычитается из всех нижележащих строк, является последовательным процессом, который начинается с нулевой строки и заканчивается последней. И процесс выбора текущей строки последовательно пересекает все полосы, обрабатываемые разными процессами. Выше лежащие строки от текущей обрабатывать не нужно. В обратном ходе наоборот. Поэтому здесь в параллельном блоке автоматическое распределение витков цикла с помощью директивы `#pragma omp for` не подходит. В данном алгоритме в параллельном блоке применено "ручное" распределение работ по параллельным процессам. При этом каждый параллельный процесс динамически вычисляет номера "своих" строк для их обработки на каждом витке циклов (прямого и обратного) (см. рис. 8.12). На рис. 8.12 схематично показан текущий момент приведения матрицы коэффициентов к диагональному виду (прямой ход). На этом рисунке ломаная линия обозначает текущую границу, слева от которой стоят

нулевые элементы, с права – не нулевые элементы. Т.е. нулевой и первый процессы свои полосы уже привели к диагональному виду, второй процесс приводит к диагональному виду только часть своих строк, третий процесс приводит к диагональному виду все свои строки. Деление элементов текущей строки на диагональный коэффициент осуществляет один процесс - master. Поэтому доступ к текущей строке другими параллельными процессами, для вычитания из "своих" строк, синхронизирован.

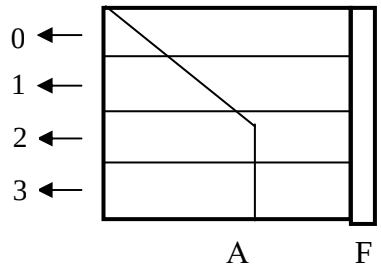


Рис. 8.12. Решение СЛАУ методом Гаусса. Полосами указаны части исходных данных, обрабатываемых разными процессами (p=4).

Текст программы, реализующий алгоритм, приведен ниже.

```
/* Пример программы на языке C (с OpenMP) решения СЛАУ методом Гаусса. */
/* Здесь в примере исходная матрица генерируется в программе */

#include<omp.h>
#include<sys/time.h>
#include<stdio.h>
```

```
#define M 1000
#define N 1000
/* Задаем исходную матрицу коэффициентов. */

static double MA[N][M+1], MAD, R;

int main()
  { int size, rank, i, j, v, k, Np, uk, dn, dk;
    struct timeval tv1, tv2, tv3, tv4;
    long int dt1,dt2;    /* Для засечения
времени */

/* Генерация исходной матрицы МA  */

    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if(i == j)
              MA[i][j] = 2.0;
            else
              MA[i][j] = 1.0;
          }
        MA[i][M] = 1.0*(M)+1.0;
      }
/* Заказываем количество параллельных процессов
*/

     omp_set_num_threads(4);
/* Задаем параллельный блок из (4-х) процессов.
Перечисленные переменные должны быть
обязательно приватным во всех процессах*/

#pragma omp parallel
private(i,j,k,rank,Np,uk,dn,dk,tv1,tv2)
```

```
  {
    size = omp_get_num_threads();
    Np = M/size;
    rank = omp_get_thread_num();
    uk = Np * rank;

   gettimeofday(&tv1, NULL);
/* Прямой ход */

    for(k = 0; k < M; k++)
      {
/* На каждом шаге цикла вычисляем полосы строк (dn-начало, dk-конец полосы), обрабатываемых каждым процессом */
      dn = (uk * (k < uk)) + ((k + 1) * (k >= uk));
      dk = uk + Np - 1;
/* Синронизация процессов */
#pragma omp barrier
/* Деление на коэффициент элементов текущей строки процессом master */
#pragma omp master
         { MAD = 1.0/MA[k][k];
           for(j = M; j >= k; j--)
            MA[k][j] *= MAD;
         }
/* Синронизация процессов для того что бы master успел обработать сроку */
#pragma omp barrier
/* Обработка строк параллельными процессами */
       for(i = dn; i <= dk; i++)
        { for(j = M; j >= k; j--)
```

```
            MA[i][j] -= MA[i][k]*MA[k][j];
          }
        }
/* Обратный ход */
      uk = (M - 1 - Np * (size - rank - 1));
      for(k = M-1; k >= 0; k--)
       {
/* На каждом шаге цикла вычисляем полосы строк
(dn-начало, dk-конец полосы), обрабатываемых
каждым процессом */
      dn = ((uk * (k > uk)) + ((k-1) * (k <=
uk)));
      dk = uk - Np + 1;
/*  Синронизация процессов, т.к. получение
корней – процесс последовательный */
#pragma omp barrier
/* Обработка строк параллельными процессами */

        for(i = dn; i >= dk; i--)
          MA[i][M] -= MA[k][M]*MA[i][k];
        }
/* Каждый процесс выводит время решения */
     gettimeofday(&tv2, NULL);
  dt1 = (tv2.tv_sec - tv1.tv_sec)*1000000+
                  tv2.tv_usec-tv1.tv_usec;
  printf(" rank1= %d Time= %ld\n",rank,dt1);
  printf(" rank1(0)= %d %f %f %f %f\n",rank,
              MA[rank][M],MA[rank+1][M],
              MA[rank+2][M],MA[rank+3][M]);
 } /* Конец параллельного блока */
 return(0);
}
```

8.3.3. Параллельный алгоритм решения СЛАУ методом сопряженных градиентов

Все формулы, по которым решается данная задача, приведены ранее. Рассматриваемая здесь схема распараллеливания алгоритма – распараллеливание циклов (SPMD-модель вычислений), т.е. *распараллеливание по данным*. Каждый параллельно исполняемый процесс обрабатывает некоторую часть назначенных ему данных аналогично, как и в MPI. Распараллеливается внешний цикл, переменная, которого указывает на строки матрицы A и элементы векторов F, X, R, и Z. Поэтому эти данные можно представить условно разрезанными, как показано на рис. 8.13, на p горизонтальных полос (p – количество параллельных процессов). На рис. 8.13 схематично показаны исходные данные задачи. Полосами указаны части исходных данных, параллельно обрабатываемых разными процессами. Такая схема определяется наличием общей памяти и независимостью выполнения циклов.

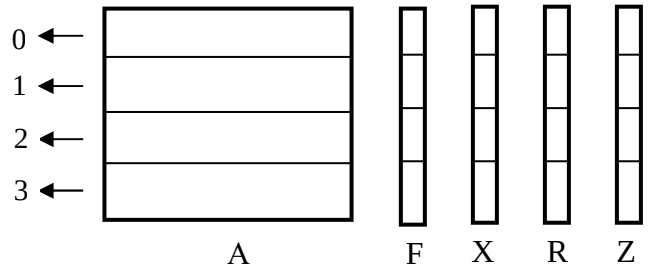


Рис. 8.13. Решение СЛАУ методом Сопряженных градиентов. Полосами указаны части исходных данных, обрабатываемых разными процессами (p=4).

Текст программы, реализующий алгоритм, приведен ниже.

```
/* Пример программы на языке C (с OpenMP) решения СЛАУ методом Сопряженных градиентов. */
/* Здесь в примере исходная матрица генерируется в программе */
#include<stdio.h>
#include<omp.h>
#include<sys/time.h>

#define M 1000

#define E 0.00001
/* Задание итерационного шага */
#define T 0.001
/* Исходные данные задаются статически */
static double A[M][M], F[M], Xk[M], Zk[M];
static double Rk[M], Sz[M], alf, bet, mf;
static double Spr, Spr1, Spz;

int main()
  { int i, j, v, size, Np, it;
    struct timeval tv1, tv2;
    long int dt1;

    mf=0;
    it = 0;


/* Генерация исходной матрицы коэффициентов и правых частей уравнений */
    for(i = 0; i < M; i++)
      { for(j = 0; j < M; j++)
          { if(i == j)
              A[i][j] = 2.0;
            else
```

```
            A[i][j] = 1.0;
          }
        F[i] = M + 1;
      mf += F[i]*F[i];
       }
/* Вычисление нормы  ||f|| */
    mf = sqrt(mf);
/* Задание начального приближения решений */
    for(i = 0; i < M; i++)
       { Xk[i] = 0.2;
         Sz[i]=0;
       }
/* Задание начальных значений векторов невязки
и сопряженного направления */
    for(i = 0; i < M; i++)
      { for(j = 0; j < M; j++)
          Sz[i] += A[i][j] * Xk[j];
      Rk[i] = F[i] - Sz[i];
      Zk[i] = Rk[i];
       }
/* Засечение времени */
    gettimeofday(&tv1,NULL);
/*  Для  решения  задачи  запрашивается  4-е
процесса */
    omp_set_num_threads(4);
/* Основной цикл */
    do
      {
        Spz = 0;
      Spr = 0;
        Spr1 = 0;
```

```
/* Начало параллельного блока праграммы */
#pragma omp parallel private(size,i)
    {  size = omp_get_num_threads();
       Np = M/size;
/*   Распараллеливание   цикла,   вычисляющего
числитель и знаменатель коэф. αk */
#pragma omp for schedule(static,Np) private(j)
reduction(+:Spz,Spr)
        for(i = 0; i < M; i++)
          { for(Sz[i]=0, j = 0; j < M; j++)
              Sz[i] += A[i][j] * Zk[j];
            Spz += Sz[i] * Zk[i];
            Spr += Rk[i] * Rk[i];
          }
/* Вычисление коэффициента αk */
#pragma omp critical
        alf = Spr/Spz;
/*   Распараллеливание   цикла,   вычисляющего
вектора решений и невязки */
#pragma omp for schedule(static,Np)
reduction(+:Spr1)
        for(i = 0; i < M; i++)
          { Xk[i] += alf*Zk[i];
            Rk[i] -= alf*Sz[i];
            Spr1 += Rk[i]*Rk[i];
          }
/* Вычисление коэффициента βk */
#pragma omp critical
        bet = Spr1/Spr;
/* Распараллеливание цикла, вычисляющего вектор
сопр. направления */
```

```
#pragma omp for schedule(static,Np)
        for(i = 0; i < M; i++)
          Zk[i] = Rk[i] + bet*Zk[i];


   }     /* Конец параллельного блока программы
*/
  it++;
  }
 while(sqrt(Spr1)/mf  >  E);     /*  Проверка  на
точность */


/* Засечение времени и вывод его значения */
    gettimeofday(&tv2,NULL);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 +
                     tv2.tv_usec - tv1.tv_usec;
    printf(" Time = %ld it=%d\n", dt1,it);
/*  Вывод  значений  первых  восьми  корней  для
контроля */
    printf("  %f %f %f %f %f %f %f %f\n",Xk[0],

Xk[1],Xk[2],Xk[3],Xk[4],Xk[5],Xk[6],Xk[7]);

    return(0);
  }
```

ОСНОВНАЯ РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. А.П.Ершов. *Вычислимость в произвольных областях и базисах*. Семиотика и информатика, вып.19, стр.3-58, 1982.
2. А.И.Мальцев. *Алгоритмы и рекурсивные функции*, Наука, 1986.
3. Х.Роджерс. *Теория рекурсивных функций и эффективная вычислимость*. Мир, Москва, 1972. (Hartley Rogers. Jr, *Theory of Recursive Functions and Effective Computability*, Mc Graw - Hill, 1967).
4. В.А.Успенский, А.Л.Семенов. *Теория алгоритмов: основные открытия и приложения*, Наука, 1987, 288 стр.
5. В.А.Вальковский, В.Э.Малышкин. *Синтез параллельных программ и систем на вычислительных моделях*. - Наука, Сибирское отделение, 1988, 128 стр.
6. В.Котов. *Сети Петри*. Наука, 1990.
7. С.М.Ачасова, О.Л.Бандман. *Корректность параллельных вычислительных процессов*. Наука, 1990.
8. Дж. Питерсон. *Теория сетей Петри и моделирование систем*. Мир, 1984 (J.Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, inc. 1981).
9. Ч.Хоар. Взаимодействующие последовательные процессы, Мир, 1989, 264 стр.
10. В.Анисимов. *Программирование распределенных вычислительных систем*. Под редакцией В.Е.Котова. Системная информатика, выпуск 3, Наука, 1993, стр. 210-247.
11. В.Н.Сачков. *Комбинаторные методы дискретной математики*, Наука, 1977, 317 стр.
12. Х.Пападимитриу, К.Стайглиц. *Комбинаторная оптимизация. Алгоритмы и сложность*. Москва, Мир, 1985
13. М.Гэри, Д.Джонсон. Вычислительные машины и труднорешаемые задачи. – М., Мир, 1982, - 416 с.
14. Ю.А.Березин, В.А.Вшивков. Метод частиц в разреженной плазме. Наука, Новосибирск. 1980.
15. Корнеев В.В, Киселев А. *Современные микропроцессоры*, 3-е издание. Санк-Петербург, БХВ-Петербург. 2003.- 440 с.

16. Таненбаум Э. *Архитектура компьютеров*. СПб.:Питер, 2002. -704с.
17. Корнеев В.В. *Параллельные вычислительные системы*. М.: Москва, 1999.- 312 с.
18. Цилькер Б. Я., Орлов С.А. *Организация ЭВМ и сетей*. СПб.: Питер, 2004. 668 с.
19. Пятьсот самых мощных компьютеров мира [http://www.top500.org].
20. В.Д.Корнеев. Параллельное программирование в MPI. – Новосибирск, ИВМ и МГ СО РАН, 2002г., 215 стр..
21. www.openmp.org
22. В.В.Воеводин, Вл.В. Воеводин. Параллельные вычисления. БХВ – Петербург2002. – 609с.
23. Г.Р.Эндрюс. Основы многопоточного параллельного и распределенного программирования. –М.: Изд. Дом Вильямс, 2003. – 330 с.

СЛОВАРЬ ТЕРМИНОВ

- *алфавит* - конечное множество символов *A=(a,b,...,c).*
- *алгоритмически вычислимые функции* - функции, вычисляемые некоторым алгоритмом.
- *асинхронная программа* (А-программа) - это конечное множество А-блоков $\{A_k | k \in \{1,2,...,m\}\}$ определенных над информационной и управляющей памятями.
- *ассоциативность кэш-памяти* – количество блоков оперативной памяти, записанные в одной строке кэш-памяти
- *буксование кэш-памяти* – это явление, вызванное последовательными обращениями к нескольким элементам, отстоящим в памяти на величину, кратную размеру кэш-памяти.
- *буфер адресов перехода* представляет собой кэш-память, которая хранит исполнительные адреса

нескольких последних команд перехода, для которых переход имел место.

- *буфер быстрого преобразования адресов* представляет собой кэш-память, которая хранит физические адреса команд и данных, к которым обращались в последнее время.
- *взаимное исключение* - допуск к ресурсу только одного процесса из множества процессов, претендующих на него.
- *виртуальный компьютер* – программно реализуемый компьютер.
- *виртуальные топологии* – программно реализуемая топология связей между компьютерами.
- *время доступа к памяти* это время между двумя последовательными операциями чтения/записи, которые выполняются по случайным адресам.
- *детерминированность алгоритма* - система величин, полученная в какой-то (не начальный)

момент времени, однозначно определённая системой величин, полученных в предшествующие моменты времени. Правильная программа, исполняясь несколько раз с одними и теми же входными данными, должна всегда выработать один и тот же результат.

- *динамические сети* – это сети, в которых любой вход может быть соединен с любым другим входом или входами.
- *дискретность алгоритма* - процесс последовательного построения величин, идущий в дискретном времени таким образом, что в начальный момент задается исходная конечная система величин, а в каждый следующий момент новая конечная система величин получается по определенному закону (программе) из системы величин, имеющихся в предыдущий момент времени.

- *длина слова* - количество символов в слове.
- *дозахват* ресурса - запрос дополнительной порции того же или другого ресурса без освобождения уже захваченных ресурсов.
- *значение* - элемент из области интерпретации, сопоставленный символу.
- *иерархическая память* – это память, представленная в виде нескольких уровней памяти, *которые* характеризуются разным временем доступа, объемом и стоимостью.
- *интерпретация* - отображение, сопоставляющее значения (элементы из области интерпретации) предметным и функциональным символам формальной системы.
- *интерливинг (чередование) адресов* - разделение адресного пространства памяти на банки таким образом, чтобы последовательные адреса находились в разных банках.

- *когерентность данных* – это идентичность данных кэш-памятей с данными основной памяти.
- *компьютеры с полным набором команд* – это компьютеры, которые характеризуются большим числом машинных команд и методов адресации, небольшой регистровый файл (8-12). Команды имеют двухадресный формат (источник и адрес результата) и переменную длину. Основные команды обмена типа «регистр-память», «память-регистр», «регистр-регистр». Команды реализованы в виде микропрограмм.
- *компьютеры с сокращенным набором команд* – это компьютеры с большим регистровым файлом, и небольшим набором команд (не более 128), Большинство команда выполняется за один машинный такт. Все команды реализуются аппаратно, имеют фиксированную длину, фиксированный формат (трехадресный) и

простые методы адресации. Для доступа к памяти введены две специальные команды обращения к памяти: *загрузка (load)* и *запись (store)*. Более того, все операции, кроме загрузки и записи, имеют тип «регистр-регистр».

- *конвейеризация* – это техника, в результате которой выполнение команды разбивается на несколько этапов. Каждый этап выполняется на своем логическом устройстве (ступени). Все ступени соединяются последовательно. В результате выполнение команды сводится к ее продвижению по конвейеру по мере освобождения последующих ступеней.
- *конкатенация (произведение) слов* - если s_1 и s_2 - слова, то слово $s_1 s_2$ называется конкатенацией слов s_1 и s_2.
- *кэш-память* – это быстрая статическая память небольшого размера, содержит копии команд и данных из оперативной памяти.

- *латентность канала связи* — время самого простого взаимодействия (обычно это передача одного машинного слова (2 байта) между узлами).
- *латентность конвейера* – время выполнения *первой команды*.
- *массовость алгоритма* - начальная система величин может выбираться из некоторого потенциально бесконечного множества.
- *многоядерный процессор* – это процессор с несколькими процессорными ядрами в одном или нескольких кристаллах.
- *МОП* – модель с общей памятью.
- *МПС* – модель передачи сообщений.
- *МПД* – модель параллелизма по данным.
- *мультикомпьютер* - несколько процессорных узлов, каждый из которых является полным компьютером. Процессорные узлы соединены коммуникационной сетью (средой). В общем

случае, процессорный узел мультикомпьютера в свою очередь может быть монопроцессором, мультипроцессором или мультикомпьютером.

- *мультимедийные команды* – это команды параллельной обработки данных в режиме «одна команда, много данных».
- *мультипроцессор* - несколько процессорных узлов над общей разделяемой памятью.
- *направленность алгоритма* - если способ получения последующей величины из какой-нибудь заданной величины не дает результата, то должно быть указано, что надо считать результатом алгоритма.
- *непроцедурность* - свойство представления алгоритма, характеризующее множество реализаций алгоритма.
- *операторы* - схемы конструирования новых функций

- *оператор минимизации* - третий из трех операторов конструирования новых вычислимых функций из простейших.
- *оператор примитивной рекурсии* - второй из трех операторов конструирования новых вычислимых функций из простейших.
- *оператор суперпозиции* (*подстановка*) - первый из трех операторов конструирования новых вычислимых функций из простейших.
- *операция a* - преобразователь, вычисляющий функцию f_α, значение которой есть результат выполнения преобразования (выполнение операции) *a*.
- *параллельное* представление алгоритма *A* - множество всех реализаций *A* содержит более одного элемента.
- *пиковая производительность процессора* определяется количеством операций,

выполняемых в единицу времени всеми функциональными устройствами компьютера.

- *последовательная программа* - программа для монопроцессора.
- *потоковое управление* - часть ограничений управления, определенных информационной зависимостью между операциями.
- *предметные символы* (*предметные переменные*)- изображают переменные
- *предотвращение дедлоков* - планирование ресурсов, при котором дедлок не может возникнуть.
- *предсказание перехода* – это техника, позволяющая выполнять выборку и декодирование потока команд по одной из ветвей, не дожидаясь проверки самого условия перехода.
- *представление* алгоритма - набор *S=(X, F, C, M)*, где *X={x, y,...,z}* - конечное множество

переменных, $F = \{a, b, \ldots , c\}$ - конечное множество операций, *C*- управление, *M* - распределение ресурсов.

- *преодоление дедлоков* – управление ресурсами, прикотором дедлок может возникнуть и существуют алгоритмы обнаружения дедлока и выхода из дедлока.
- *пропускная способность* или полоса пропускания канала связи – скорость передачи единицы информации в единицу времени (Мбайт/сек).
- *простейшие рекурсивные функции (вычислимы по определению):* а).функция следования $s(x)= x+1;$ б).нулевая функция $o(x)= 0;$ в).функции выбора $I_m^n(x_1, x_2, \ldots , x_n)= x_m,\ m < n$
- *процесс (последовательный процесс)* - исполняющаяся программа (*программа процесса*) со своими входными данными. Одна и та же программа с разными входными данными определяет разные процессы.

- *процессор с длинным командным словом* – это процессор, в котором параллельная обработка указывается явно в специально отведенных полях команды. Выявление одновременно выполняемых команд обеспечивает компилятор.
- *прямое управление* - ограничения управления, не попавшие в потоковые, связаны обычно с распределением ресурсов и оптимизацией выполнения программ
- *разделяемый ресурс* - ресурс, который одновременно могут использовать несколько процессов
- *распределение ресурсов мультикомпьютера* - функция, задающая отображение множеств переменных и операций алгоритма в физические устройства многопроцессорной вычислительной системы
- *реализация* алгоритма *A*, представленного в форме *S* - это выполнение операций алгоритма в

некотором произвольном допустимом порядке, который не противоречит управлению *C*.

- *семафор* - переменная особого типа, которая после инициализации начального значения доступна только посредством семафорных операций *P* и *V*.
- *сеть Петри* - двудольный ориентированный граф специального вида для описания поведения системы параллельно протекающих и взаимодействующих процессов.
- *состояние дедлока* - возникает в системе процессов, когда запросы ресурсов в системе не могут быть удовлетворены и система останавливается
- *спекулятивное исполнение команд*— это исполнение команд выбранной ветви до завершения проверки условия перехода.

- *суперскалярный процессор* — это процессор, в котором одновременное выполнение несколько команд достигается аппаратными средствами.
- *статические сети* – это сети, которые имеют жестко зафиксированные соединения.
- *тезис Черча* – утверждение о том, что класс всех вычислимых функций совпадает с классом частично рекурсивных функций.
- *термы* (*функциональные термы*) - слова особого вида, записанные в функциональном алфавите.
- *управление* - множество ограничений на порядок выполнения операций
- *функциональные символы* - изображают функции.
- *частично рекурсивная функция* – функция, построенная конечным числом применений операторов суперпозиции, примитивной рекурсии и минимизации.

- *элементарность шагов алгоритма* - закон получения последующей системы величин из предшествующей должен быть простым и локальным.
- *MPI (Message Passing Interface)* – язык параллельного программирования (реализован в виде библиотеки), ориентированный на вычислительные системы с распределенной памятью.
- *OpenMP* (OpenMP ARB (ARchitecture Board) – организация, в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения) – язык параллельного программирования (реализован в виде директив препроцессора), ориентированный на вычислительные системы с общей памятью.
- *3D область* – область трехмерного вычислительного пространства.

- *MPMD (Multiple program - Multiple Data)* – модель программирования. Параллельная программа представляет собой совокупность автономных процессов, функционирующих под управлением своих собственных программ и взаимодействующих посредством стандартного набора библиотечных процедур для передачи и приема сообщений.
- *SPMD (Single program - Multiple Data)* – модель вычислений. Все процессы исполняют в общем случае различные ветви одной и той же программы.