

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

Д. Л. Головашкин

**Современные методы и алгоритмы
решения сложных задач
на суперкомпьютерах**

Электронное учебное пособие

САМАРА
2010

УДК 681.324.006.3

ББК 39.65

Автор: **Головашкин Димитрий Львович**

Рецензенты: к.ф.-м.н., доцент С.И.Харитонов,

к.т.н. , доцент С.Б. Попов

В настоящей работе излагаются методики построения параллельных алгоритмов, вычисления по которым могут быть реализованы на системах с мультипроцессорной архитектурой. Рассматриваются особенности, связанные с декомпозицией исходного численного метода, организацией коммуникаций и объединением задач параллельного алгоритма. Приведены примеры алгоритмов для задач математической физики, решаемых посредством теории разностных схем.

Учебное пособие рекомендуется для магистрантов по курсу лекций «Современные методы и алгоритмы решения сложных задач на суперкомпьютерах» в рамках магистерской программы «Технологии параллельного программирования и суперкомпьютеринг» по направлению 01400.68 – «Прикладная математика и информатика» и может быть полезно при выполнении курсовых работ, дипломных проектов и подготовке к экзаменам.

Подготовлено на кафедре прикладной математики.

© Самарский государственный
аэрокосмический университет, 2010

ОГЛАВЛЕНИЕ

Введение	4
1 Распараллеливание последовательных программ	7
1.1 Распараллеливание ациклических участков программ	7
1.2 Распараллеливание ациклических участков программ	15
1.3 Распараллеливание выражений	27
2 Синтез параллельных алгоритмов	34
2.1 Понятие параллельного алгоритма	34
2.1 Восходящий подход к синтезу параллельных алгоритмов	36
2.3 Нисходящий подход к синтезу параллельных алгоритмов	59
2.4 Характеристики параллельных вычислительных процессов и алгоритмов	65
3. Параллельные алгоритмы решения разностных уравнений	74
3.1 Параллельные алгоритмы решения сеточных уравнений явных разностных схем	74
3.2 Параллельные алгоритмы решения трехдиагональных сеточных уравнений неявных разностных схем	84
Заключение	103
Литература	104

Введение

Способность производить вычисления является не только необходимым навыком для большинства людей, но и фундаментальным признаком любой цивилизации. Вряд ли это утверждение верно для параллельных вычислений в приложении к цивилизациям древности, но в современном мире параллельные вычисления прочно обосновались в научных расчетах и начинают распространяться в прикладные области (базы данных, декодирование кодирование информации т.д.). Разрабатываются специальные параллельные алгоритмы, по ним пишутся параллельные программы, исполняемые на соответствующих вычислительных системах, допускающих параллельные вычисления. Обычно повышенный интерес к параллельной тематике связывают с совершенствованием вычислительной техники и сетевых технологий, хотя идея одновременно исполнять независимые операции над числами была воплощена еще до создания ЭВМ, способных на это.

Вот как об этом пишет Владимир Валентинович Воеводин в курсе лекций "Параллельная обработка данных" /1/. "Кстати, пионером в параллельной обработке потоков данных был академик А.А.Самарский, выполнявший в начале 50-х годов расчеты, необходимые для моделирования ядерных взрывов. Самарский решил эту задачу, посадив несколько десятков барышень с арифмометрами за столы. Барышни передавали данные друг другу просто на словах и откладывали необходимые цифры на арифмометрах. Таким образом, в частности, была рассчитана эволюция взрывной волны. Работы было много, барышни уставали, а Александр Андреевич ходил между ними и подбадривал. Это, можно сказать, и была первая параллельная система. Хотя расчеты водородной бомбы были мастерски проведены, точность их была очень низкая, потому что узлов в используемой сетке было мало, а время счета получалось слишком большим".

Развитие вычислительных систем в значительной мере задает темп научно-технической революции, поэтому изучение архитектуры современных вычислительных комплексов и их программного обеспечения весьма важно. Возрастающая популярность вычислений на графических процессорах требует глубоких знаний архитектуры и программного обеспечения, отличных от традиционных. На горизонте маячит образ квантовых компьютеров, архитектура которых кардинальным образом отличается от всех современных. Соответственно программные комплексы для таких ЭВМ также окажутся весьма специфическими. Но высокая скорость такого развития требует постоянного пополнения знаний в этой области, что не всегда удобно для университетского курса, который призван давать фундаментальные знания, опираясь на которые специалист будет способен в дальнейшем отслеживать изменения в соответствующей области.

В силу изложенного очевидна удачность названия курса “Современные методы и алгоритмы решения сложных задач на суперкомпьютерах”. Конечно, и алгоритмы и методы тоже претерпевают изменения, которые, однако, в большей степени связаны с развитием математических методов решения задач вычислительной физики и в значительно меньшей степени зависят от архитектуры ЭВМ и программных комплексов.

На практике, особенно часто у молодых специалистов, встречаются ситуации, когда предметная область исследований еще не знакома, а острая необходимость в параллельной программе серьезно ограничивает время на ее создание. Тогда представляется разумным, не тратя времени на разработку параллельного алгоритма из "физики" задачи, использовать имеющуюся последовательную программу для синтеза необходимого алгоритма. Последовательная программа разбивается на циклические и ациклические фрагменты, к которым применяются известные методики распараллеливания. Проведенная ранее формализация таких методик позволяет на сегодняшний день проводить распараллеливание в автоматическом режиме, поручив его компьютеру. Первая глава пособия и посвящена методикам

распараллеливания и их адаптации к конкретным ситуациям из вычислительной практики.

Во второй главе читателю предлагаются приемы распараллеливания параллельных алгоритмов, которые уже понятны исследователю. В качестве инструментов распараллеливания используются два подхода, соответствующие известной гегелевской паре “тезис”, “антитезис”. Автор пособия рассчитывает на получение диалектического “синтеза” пытливым исследователем, овладевшим этими инструментами.

Заключает изложение материал третьей главы, целиком посвященный разнообразным параллельным алгоритмам решения задач математической физики посредством разностных схем. Рассмотрены как явные, так и неявные схемы, записанные на сеточных областях различной размерности. Приведенные результаты свидетельствуют о высокой эффективности рекомендуемых алгоритмов.

Для понимания материала, изложенного в пособии, необходимо прочное владение знаниями из курса дискретной математики. С первой до последней страницы данного издания активно используется аппарат теории множеств. Понятия бинарных отношений и отображений. Сведения из теории графов и элементы математического анализа. Теория разностных схем необходима для понимания заключительной главы пособия.

ГЛАВА 1 РАСПАРАЛЛЕЛИВАНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ПРОГРАММ

При написании параллельного алгоритма полезно основываться на существующем последовательном алгоритме. Но как быть в случае, когда для понимания последовательного алгоритма необходимы специальные знания? Например, прикладному математику ставится задача по написанию параллельного алгоритма для расчетов в области молекулярной биологии. Последовательный алгоритм будет понят не скоро, соответственно не скоро удастся создать из него параллельный. Разумнее воспользоваться известной последовательной программой, сконструировав параллельный алгоритм из нее (распараллелив). Сама по себе последовательная программа оперирует с областью памяти, порождая вычисления. Для ее анализа не нужны знания в области микробиологии.

В последовательной программе отдельно будем рассматривать ациклические и циклические участки, применяя для их распараллеливания различные приемы.

1.1 Распараллеливание ациклических участков программ.

Перед распараллеливанием ациклического участка параллельной программы перепишем его с использованием только двух типов операторов: присваивания и условного перехода. Возможно, от этого пострадает лаконичность кода, но такая стандартизация упрощает распараллеливание, в дальнейшем удобнее иметь дело всего с двумя типами операторов.

По новому коду построим ориентированный граф /2/, вершинами которого являются операторы программы, а дуги задаются отношением непосредственного следования на множестве операторов. Для последовательной программы это отношение определяется порядком следования операторов в программе и условными переходами. Различают два типа вершин такого графа. Распознаватели соответствуют операторам условного перехода и обозначаются на графе кружками, преобразователи - операторами присваивания и обозначаются прямоугольниками. Одному

преобразователю может соответствовать несколько операторов присваивания, одному распознавателю только один оператор условного перехода. Очевидно, из преобразователя будет вести максимум одна дуга, из распознавателя две, соответствующие истинности условия (такую дугу помещают цифрой "1") и его ложности (помечают цифрой "0"). Вершину, в которую не ведут дуги, назовем входной вершиной графа, вершину, из которой не ведут дуги - выходной, а сам граф - стандартным.

Проиллюстрируем изложенное примером. Следующий фрагмент программы

```

A: ввод(x,y);
B: l:=x;
C: h:=y;
D: v:=c+y;
E: z:=c+x;
P: если x>y то на F иначе на G;
F: h:=x; l:=y;
G: распечатка (min(z,v), max(z,v));
H: печать (l,h);

```

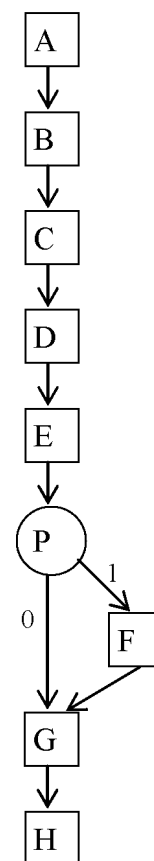


Рис. 1.1 Стандартный граф программы.

по введенным относительным индексам x и y массива вычисляет абсолютные индексы v и z , используя базу массива c , распечатывает участок массива между абсолютными индексами и выводит упорядоченные относительные индексы. На рис. 1.1. представлен стандартный граф этого участка программы. На нем вершина A будет входной, вершина H выходной.

От стандартного графа перейдем к графу зависимостей. Вершины графа зависимостей совпадают с вершинами стандартного графа, дуги задаются тремя бинарными отношениями, которые в данном случае принято называть зависимостями. Рассмотрим эти отношения.

а. Информационная зависимость.

Говорят, что вершина В информационно зависит от вершины А, если А и В находятся на стандартном графе на одном пути из входной вершины в выходную, причем А предшествует В, и хотя бы один из операторов вершины А формирует значение переменной x , которой пользуется, но значение которой не изменяет, хотя бы один оператор из вершины В ($In(B) \cap Out(A) \neq \emptyset$), при этом на рассматриваемом пути между А и В отсутствует вершина С, хотя бы один оператор которой меняет значение x . $In(B)$ есть множество переменных, находящихся в правой части группы операторов присваивания из вершины В, если вершина В - преобразователь и входящих в логическое условие, если В-распознаватель; $Out(A)$ - множество переменных, находящихся в левой части группы операторов присваивания из А. Вершина, от которой информационно зависят, всегда преобразователь, вершина, которая информационно зависит, может являться как преобразователем, так и распознавателем.

б. Логическая зависимость.

Вершина В логически зависит от вершины А, если они находятся на одном пути стандартного графа от входной вершины в выходную (А предшествует В) и существует другой путь из входной вершины в выходную, совпадающий с первым до вершины А включительно и отличающийся от него далее, причем второй путь уже не содержит вершины В. Вершина, от которой логически зависят, всегда распознаватель, вершина, которая логически зависит, может быть как распознавателем, так и преобразователем.

в. Конкуренционная зависимость.

Если вершины А и В находятся на одном пути стандартного графа из входной вершины в выходную, причем на этом пути А предшествует В, и $Out(A) \cap Out(B) \neq \emptyset$, то В конкуренционно зависит от А. В отношение конкуренционной зависимости вступают только преобразователи.

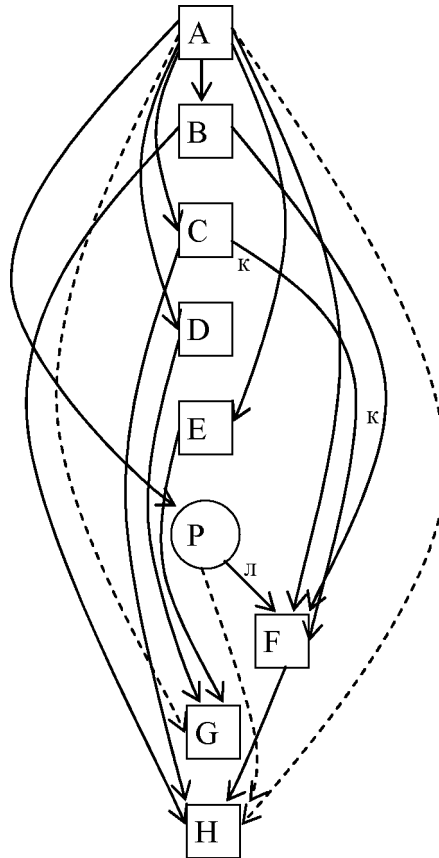


Рис. 1.2 Граф зависимостей. Дуги без маркировки соответствуют информационной зависимости, дуга с буквой "л" логической зависимости, дуги с буквой "к" конкуренционной зависимости, пунктирные дуги транзитивному замыканию всех зависимостей.

Добавив дуги, соответствующие транзитивному замыканию всех зависимостей получим граф зависимостей. На рис. 1.2 дуги графа зависимостей маркированы по-разному только для ясности изложения, в дальнейшем нас не будет интересовать тип зависимостей, важно лишь наличие или отсутствие дуги (зависимости) из одной вершины в другую, которая обозначается как $Dep(A,B)$: вершина B зависит от вершины A. Далее, говоря о пути на графе, будем иметь в виду стандартный граф, говоря о зависимости - граф зависимостей.

Следующий этап по распараллеливанию ациклического фрагмента последовательной программы - построение ярусно-параллельной формы (ЯПФ). В первый ярус ЯПФ поместим все независимые вершины, в каждый следующий ярус все те вершины, что зависят только от вершин предыдущих ярусов (рис. 1.3).

ярус I	A
ярус II	B; C; D; E; P
ярус III	F; G
ярус IV	H

Рис. 1.3 Ярусно-параллельная форма.

Операторы, соответствующие вершинам одного яруса могут выполняться одновременно в силу независимости вершин. Вершины разных ярусов тоже могут оказаться независимыми (например H не зависит от G), но их операторы одновременно выполняться не могут, так как вершина некоторого яруса обязательно будет зависеть хотя бы от одной вершины предыдущего яруса (H зависит от F).. По ЯПФ составляется параллельный алгоритм в соответствии с архитектурой вычислительной системы. В следующем алгоритме формализован процесс составления ЯПФ и написания параллельного алгоритма в допущении синхронности вычислений в широком смысле с использованием модели канал/задача. Задачи в искомом параллельном алгоритме будут состоять из последовательности вершин стандартного графа. Предполагается, что ациклический фрагмент последовательной программы уже переписан с использованием только двух типов операторов, построены стандартный граф и граф зависимостей.

шаг 1. Первый ярус ЯПФ составляется из всех вершин A таких, что

$$\forall B \neg \text{Dep}(B, A)$$

шаг 2. Предполагается, что k ярусов уже построены (рис.1.4). Если для

$$\forall A \exists_{i \leq k} \exists_{j \leq n_k} \text{ такие, что } A = A_{i,j} \text{ (все вершины распределены по ярусам)}$$

то переходим на шаг4, иначе на шаг 3.

ярус I	$A_{1,1}; A_{1,2}; \dots A_{1,n_1};$
ярус II	$A_{2,1}; A_{2,2}; \dots A_{2,n_2};$
.....	
ярус k	$A_{k,1}; A_{k,2}; \dots A_{k,n_k};$

Рис. 1.4 Произвольная ярусно-параллельная форма. n_i - число вершин i-ого яруса.

шаг 3. Заполнение $k+1$ яруса. Он заполняется вершинами, не вошедшими в предыдущие ярусы и зависящими только от вершин этих ярусов. То есть A такими, что $\forall A (Dep(B,A) \Rightarrow \exists_{i \leq k} \exists_{j \leq n_k} (A_{i,j} = B))$. Заполнив ярус переходим на шаг 2.

шаг 4. Начало построения параллельного алгоритма, состоящего из N задач. На этом и следующем шаге будем распределять вершины графа между задачами. Независимые вершины первого яруса распределим таким образом, чтобы в s -ую задачу ($s=1, N$) вошли вершины в такой последовательности:

$$A_{1,s}; A_{1,s+N}; A_{1,s+2N}; \dots A_{1,s+qN}; \text{ где } s+qN \leq n_1$$

шаг 5. Предполагается, что вершины k ярусов уже распределены по задачам. Если все ярусы исчерпаны, переходим на шаг 7, иначе на шаг 6.

шаг 6. Распределим вершины $k+1$ яруса между задачами следующим образом. К вершинам s -ой задачи будут добавлены логические функции c и вершины

$$c(A_{k+1,s}); A_{k+1,s}; c(A_{k+1,s+N}); A_{k+1,s+N}; \dots c(A_{k+1,s+qN}); A_{k+1,s+qN}; \text{ где } s+qN \leq n_{k+1}$$

Если логическая функция, предшествующая вершине, примет значение "истина", то операторы этой вершины могут исполняться, иначе управление передается на следующую логическую функцию.

Возвращаемся к шагу 5.

шаг 7. Параллельный алгоритм считается построенным.

Поясним способ задания логической функции $c(A)$ на шаге 6.

В общем случае через вершину A проходят L путей из входной вершины в выходную. Множество этих путей обозначим как $\psi = \{v^1, v^2, \dots v^L\}$. Тогда $c(A) = c^1 \vee c^2 \vee \dots c^L$, где c^l логическая функция, связанная с путем l ($l=1, L$). Пусть путь v^l от входной вершины до вершины A состоит из $A_1^l; A_2^l; \dots A_m^l$, (очевидно A_1^l - входная вершина, $A_m^l = A$), причем A зависит только от $A_1^l; A_2^l; \dots A_g^l$. В этом случае $c^l = d_1^l \wedge d_2^l \wedge \dots d_g^l$, где d_h^l ($h=1, g$) определяется как

$$\left\{ \begin{array}{l} c(A_h^1), \text{ если } A_h^1 \text{ преобразователь;} \\ c(A_h^1 = 1), \text{ если } A_h^1 \text{ распознаватель и для осуществления пути } v^1 \\ \quad \text{необходим переход по ребру "истина";} \\ c(A_h^1 = 0), \text{ если } A_h^1 \text{ распознаватель и для осуществления пути } v^1 \\ \quad \text{необходим переход по ребру "ложь".} \end{array} \right.$$

При составлении логической функции $c(A)$ иногда удастся избежать включения в нее логических функций от некоторых вершин, например $c(B)$, хотя A зависит от B . Такое возможно в трех случаях.

1. Если вершина B предшествует вершине A в одной задаче. Тогда операторы из A исполняться после операторов из B в силу того, что операторы одной задачи исполняются последовательно.
2. Если существует вершина C , предшествующая в одной задаче вершине A и в функцию $c(C)$ входит $c(A)$.
3. Если существуют вершины C и D такие, что C предшествует A и B предшествует D , каждая в своей задаче, причем в функцию $c(C)$ входит $c(D)$.

Отметим, что если в логической функции данной задачи от какой-либо вершины присутствует логическая функция некой вершины другой задачи, операторы которой еще не исполнились, то вычислительный процесс, связанный с данной задачей приостановится до тех пор, пока вычислительный процесс, заданный другой задачей не исполнит или проигнорирует операторы некой вершины.

В алгоритме логическая функция от вершины предшествует операторам вершины, что напоминает конструкцию асинхронной программы. Однако это чисто внешнее сходство, операторы каждой задачи исполняются последовательно, также последовательно формируются значения предшествующих им логических функций.

Используем изложенный алгоритм для распараллеливания приведенного участка программы на три задачи (рис.1.5).

ярус	задача 1	задача 2	задача 3
I	A		
II	B	$c(A); C$	$c(A); D$
	E	P	
III	$c(P=1); F$	$c(D) \wedge c(E); G$	
IV	$c(P=0) \vee c(P=1); H$		

Рис. 1.5. Распределение вершин стандартного графа между задачами параллельного алгоритма.

Вершина A входная, ей не может предшествовать логическая функция. Вершины B и C зависят от A, но следуют за A в одной задаче, следовательно они не нуждаются в логических функциях. Вершины C и D зависят от A и находятся с A в разных задачах, им соответствует функция $c(A)$. Вершина P зависит от A, но предшествующая ей во второй задаче вершина C тоже зависит от A, значит перед P логическая функция не обязательна. Вершина F зависит от A, B, C и P; A и B предшествуют ей в одной задаче, C предшествует во второй задаче P, таким образом в логическую функцию для F войдет только зависимость от P. G зависит от A, D и E; при чем зависимость от A в этой задаче уже присутствует, значит перед G будет конъюнкция $c(D) \wedge c(E)$. H зависит от A, B, C, P и F; в логическую функцию войдет только дизъюнкция $c(P=0) \vee c(P=1)$, ведь H следует за P в двух различных путях на стандартном графе.

Предъявим два требования к параллельному алгоритму, построенному по ациклическому фрагменту программы.

1. Вершины должны распределяться между ветвями как можно более равномерно.
2. Логические функции, предшествующие операторам должны иметь как можно более простой вид.

Изменив содержание шагов 4-6 (в сторону усложнения) можно построить более оптимальный параллельный алгоритм, лучше удовлетворяющий приведенным требованиям.

Для сравнения параллельных программ, записанных по различным алгоритмам удобно пользоваться следующими определениями.

Данная программа будет не менее параллельной, чем некоторая, эквивалентная ей, если множество вычислительных процессов, порожденных данной программой содержит в качестве подмножества множество вычислительных процессов, порожденных некоторой программой.

Назовем программу максимально параллельной, если она будет не менее параллельна, чем все эквивалентные ей программы.

При распараллеливании последовательной программы естественно стремиться к синтезу такого параллельного алгоритма, по которому будет записана максимально параллельная программа.

1.2. Распараллеливание циклических участков программ.

В большинстве алгоритмов для численных методов отношение числа операций, производящихся в циклах, к общему числу операций, близко к единице. Следовательно, можно рассматривать задачу распараллеливания циклических участков программ как весьма важную. Рассмотрим участок последовательной программы, состоящий из k вложенных циклов /2/

```

for i1:=1 to n1 do
  for i2:=1 to n2 do
    .....
    for ik:=1 to nk do
      T(i1,i2, ... ,ik);

```

Программа 1.1 Вложенная циклическая конструкция общего вида.

где $T(i_1, i_2, \dots, i_k)$; -тело цикла. Каждой итерации этой циклической конструкции поставим в соответствие целочисленный вектор $(i_1, i_2, \dots, i_k)$, где $i_g = 1, n_g$ а $g = 1, k$. Пространство итераций $I = \{(i_1, i_2, \dots, i_k) : 1 \leq i_g \leq n_g, 1 \leq g \leq k\}$ состоит из таких векторов. Очевидно, между пространством итераций и множеством итераций циклической конструкции можно установить взаимно однозначное соответствие (биекцию). Наша задача - найти совокупность множеств $\{I_q\}_{q=1, s}$,

являющуюся покрытием пространства итераций ($I = \bigcup_{q=1}^s I_q$), таких, что вектора

каждого множества I_q независимы друг от друга. Для определения отношения

зависимости на пространстве итераций "развернем" циклическую конструкцию в ациклический фрагмент

$$T(1,1, \dots 1); T(1,1, \dots 2); \dots T(n_1, n_2, \dots n_k);$$

и построим для него граф зависимостей. Если хотя бы две вершины такого графа, содержащие операторы из T' и T'' , окажутся в отношении зависимости, то назовем зависимыми и соответствующие T' и T'' вектора из I . Иногда достаточно выделить только часть зависимостей, например информационную или конкуренционную. Итерации, связанные с векторами одного множества I_q , можно выполнять независимо друг от друга (параллельно). Каждое покрытие пространства I и соответствующий параллельный алгоритм характеризуется

величиной $\chi = \left(\prod_{j=1}^k n_j \right) / s$ - ускорением. Следует искать покрытие,

обеспечивающее максимальное ускорение параллельного алгоритма. Известны четыре метода решения такой задачи: метод гиперплоскостей, координат, пирамид и параллелепипедов.

1.2.1 Метод гиперплоскостей.

На рис. 1.6 представлена геометрическая интерпретация пространства итерации для циклов

for i:=1 to r do

for j:=2 to m do

for k:=2 to n do

$$x(j,k) := f(x(j-1,k), x(j+1,k), x(j,k-1), x(j,k+1));$$

Для выбранного вектора x установим зависимости от ближайших векторов a, b, c, d . Заметим, что если вектора c и d располагаются на слое $i-1$, то вектора a и b находятся уже на слое i , что связано с особенностями исполнения итераций. Через прямые $l_1: j+k=\text{const.}$ и $l_2: j+k=\text{const.}+2$, содержащие независимые итерации, проведем плоскость. Совокупность таких плоскостей, параллельных друг другу, составит искомое покрытие (это покрытие будет разбиением

пространства итераций). Плоскости являются покрытием для трехмерного пространства итераций, в общем случае говорят о гиперплоскостях.

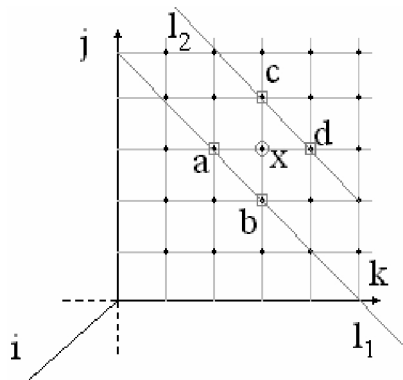


Рис. 1.6 Построение гиперплоскости для трехмерного пространства итераций (проекция на один слой по i).

цикл по i'

$i'=6$ to $2*r+m+n$

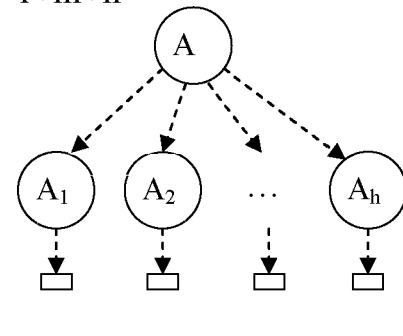


Рис. 1.7. Модель параллельного алгоритма для метода гиперплоскостей.

Наклоним координатные оси так, чтобы новая ось i' стала перпендикулярна плоскостям, содержащим независимые вектора. Тогда $i'=2i+j+k$, $j'=i$, $k'=k$. В параллельном алгоритме внешний цикл по i' , заданный в задаче A (рис. 1.7), будет перебирать гиперплоскости $I_{i'}=\{(i',j',k'): 1 \leq j' \leq r, 2 \leq k' \leq n, 2 \leq i'-2j'-k' \leq m\}$,

$i'=6, 2*r+m+n$. Вычислением тел

$$x(i'-2*j'-k',k'):=f(x(i'-2*j'-k'-1,k'),x(i'-2*j'-k'+1,k'),x(i'-2*j'-k',k'+1),x(i'-2*j'-k',k'-1)),$$

связанных с векторами одной гиперплоскости, занимаются порожденные A задачи $A_1; A_2; \dots A_h$, каждая для своей пары j',k' . Необходимость в порождении и удалении задач объясняется тем, что в разных гиперплоскостях может находиться разное количество векторов и h будет функцией от i' .

Суть метода гиперплоскостей состоит в нахождении гиперплоскостей, разбивающих в теоретико-множественном смысле пространство итераций и поворота координатных осей для организации цикла по перебору гиперплоскостей. Такой подход налагает следующие ограничения.

1. Метод не применим для простых циклов. В этом случае гиперплоскостями будут вектора пространства итераций.
2. Не допустимо изменение параметров цикла в его теле.

3. Использование параметров цикла возможно только в линейных выражениях.
4. Никакой оператор условного перехода не должен вести за тело цикла.

Алгоритм, построенный по методу гиперплоскостей, желательно использовать для вычислений над общей памятью, так как динамическое порождение задач над иначе организованной памятью крайне нежелательно.

1.2.2 Метод координат.

Построим гиперплоскости (рис. 1.8) в пространстве итераций, связанном с циклами

```

for j:=1 to 3 do
  for i:=1 to 4 do begin
     $x(i,j)=f(x(i-1,j)); y(i,j)=f_1(y(i,j-1)); u(i,j)=f_2(u(i,j-1)); p(i,j)=f_3(p(i,j-1));$ 
  end;

```

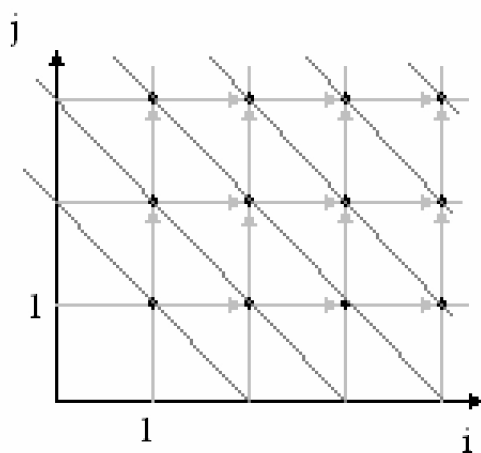


Рис. 1.8 Гиперплоскости для двумерного пространства итераций.

Стрелками изображены информационные зависимости.

цикл по j

j=1 to 3

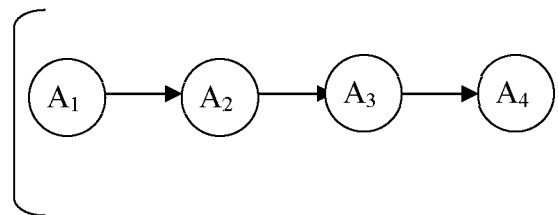


Рис. 1.9. Модель параллельного алгоритма для метода координат.

Стрелками обозначены каналы, по которым происходит синхронизация.

Изобразив пространство итераций на плоскости отметим, что для $i > 2$ и $j > 1$ каждый вектор информационно зависит от своих соседей "слева" и "снизу". Можно дополнить рис. 1.8 транзитивным замыканием информационных зависимостей (других зависимостей в циклах нет), но это не повлияет на расположение гиперплоскостей, которые для двумерного пространства будут

прямыми. Построив алгоритм, аналогичный алгоритму на рис. 1.7 получим ускорение $\chi=12/6=2$.

Метод координат также требует геометрической интерпретации пространства итераций и построения гиперплоскостей, однако вместо поворота осей он подразумевает поворот семейства гиперплоскостей перпендикулярно одной из осей. Обычно это достигается изменением порядка следования оператора цикла и введением синхронизации между задачами параллельного алгоритма. Для представленного примера параллельный алгоритм, полученный с помощью метода координат, схематично изображен на рис. 1.9. На рис. 1.10 предложен вариант изменения очередности следования операторов в теле цикла для каждой задачи.

Задача A_1 (для $i=1$)	Задача A_2 (для $i=2$)	Задача A_3 (для $i=3$)	Задача A_4 (для $i=4$)
For $j=1$ to 3 do			
$x(i,j)=f(x(i-1,j));$ —————→	$Y(i,j)=f_1(y(i,j-1));$ —————→	$y(i,j)=f_1(y(i,j-1));$ —————→	$y(i,j)=f_1(y(i,j-1));$ —————→
$y(i,j)=f_1(y(i,j-1));$ —————→	$X(i,j)=f(x(i-1,j));$ —————→	$u(i,j)=f_2(u(i,j-1));$ —————→	$u(i,j)=f_2(u(i,j-1));$ —————→
$u(i,j)=f_2(u(i,j-1));$ —————→	$U(i,j)=f_2(u(i,j-1));$ —————→	$x(i,j)=f(x(i-1,j));$ —————→	$p(i,j)=f_3(p(i,j-1));$ —————→
$p(i,j)=f_3(p(i,j-1));$ —————→	$P(i,j)=f_3(p(i,j-1));$ —————→	$p(i,j)=f_3(p(i,j-1));$ —————→	$x(i,j)=f(x(i-1,j));$ —————→

Рис. 1.10 Изменения очередности следования операторов в теле цикла для каждой задачи из рис. 1.9 Стрелки означают синхронизацию задач.

Синхронизация обеспечивает одновременное начало исполнения любого оператора тела цикла для всех задач. Одновременно начнут исполняться первые операторы задач, потом вторые и т.д. Вторая задача во втором операторе тела цикла использует значение переменной, изменяемой в первом операторе первой задачи, следовательно, одновременное исполнение этих операторов недопустимо, что и гарантирует синхронизация. Порядок исполнения операторов был изменен так, чтобы значение переменной x формировалось до

того, как оно понадобится другой задаче, а синхронизация не позволяет задачам исполнять операторы "в разнбой", "опережая" или "отставая" от других задач. Таким образом, удалось учесть информационную зависимость вектора от его левого соседа на рис. 1.8, не разворачивая координатные оси. Это дает ощутимый выигрыш в ускорении, которое составит $\chi=12/3=4$.

К достоинствам метода координат следует отнести постоянное количество задач, составляющих параллельный алгоритм, что позволяет работать и над разделенной памятью (отпадает необходимость в динамическом порождении задач). В случае использования разделенной памяти синхронизация будет сопровождаться передачей данных, в последнем примере элементов массива x .

Недостатками метода являются необходимость в изменении порядка операторов (новый порядок не всегда очевиден и его формирование может потребовать существенных изменений кода) и задание синхронизаций между задачами, которые увеличат время работы параллельной программы, составленной по такому алгоритму. Четыре ограничения, налагаемые при использовании метода гиперплоскостей действительны и для метода координат.

1.2.3. Метод пирамид.

Выделим на пространстве итераций вектора, от которых не зависит ни один вектор данного пространства, учитывая, что отношение зависимости антирефлексивно. Такие вектора назовем результирующими. Введем на пространстве итераций отношение следования. Если в "развернутой" циклической конструкции первый оператор из T' следует за последним оператором из T'' , то вектор, связанный с T' следует за вектором, связанным с T'' . Таким образом, пространство итераций упорядочено в теоретико-множественном смысле. Можно утверждать, что переменные, значения которых формируются телами, соответствующими результирующим векторам, не изменяются ни в одном теле, соответствующем любому следующему вектору. Каждому результирующему вектору поставим в соответствие

(биективное) одну задачу синтезируемого параллельного алгоритма. Этой задаче поставим в соответствие (не инъективное) все вектора из пространства итераций, от которых зависит взятый результирующий вектор. В задаче расположим операторы в соответствии с отношением следования, определенном на пространстве итераций. То есть операторы тела цикла, относящегося к результирующему вектору p , окажутся в задаче после всех остальных операторов. Им будут непосредственно предшествовать операторы тела цикла, относящегося к вектору p_1 , за которым следует результирующий вектор, при чем в пространстве итераций не должно существовать такого вектора p_2 , который следует за p_1 , и предшествует p . Первыми в задачу войдут операторы тела, связанного с независимым вектором, от которого зависят все вектора, поставленные в инъективное соответствие задаче.

Поясним изложенное примером. Рассмотрим циклическую конструкцию

```
for i:=1 to r1 do
  for j:=1 to r2 do
    T(i,j);
```

для которой результирующими векторами будут вектора $p = \{(r1, j): 1 \leq j \leq r2\}$ (рис.1.11), а вектора, от которых будет зависеть выбранный результирующий (помечен звездочкой), находятся в треугольнике между прямыми l и h .

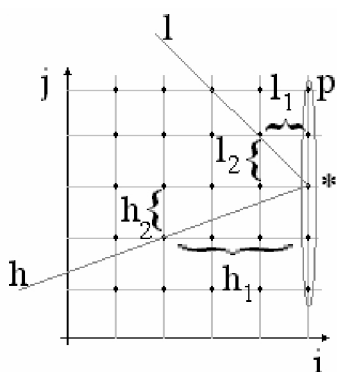


Рис. 1.11 Пространство итераций с выделенными результирующими итерациями.

цикл по i

$i=1$ to $r1$

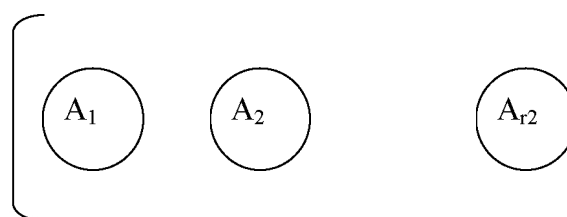


Рис. 1.12. Модель параллельного алгоритма для метода пирамид ($r2$ задач).

Параллельный алгоритм будет состоять из $r2$ задач, k -ая задача ($k=1, r2$) в цикле по i (рис. 1.12) производит следующие действия:

for $j := \max\{1, k - (r_1 - i) * \operatorname{tg}(\alpha)\}$ to $\min\{r_2, k + (r_1 - i) * \operatorname{tg}(\beta)\}$ do
 $T(i, j);$

где $\operatorname{tg}(\alpha) = h_2/h_1$, $\operatorname{tg}(\beta) = l_2/l_1$. α - острый угол между осью i и прямой l , β - острый угол между осью i и прямой h . Цикл по j пробегает все вектора с координатой i , лежащие внутри треугольника, ограниченного прямыми l и h .

Отметим, что если в теле цикла, присутствуют нелинейные выражения, оперирующие с параметрами цикла, то множество векторов, от которых зависит выбранный результирующий вектор, будет ограничиваться на плоскости не прямыми, образующими стороны треугольника, содержащего такие вектора, а кривыми линиями (если I двумерно). Так, метод пирамид допускает распараллеливание циклов, в теле которых параметры цикла входят в нелинейные выражения. Остальные три ограничения, сформулированные при описании метода гиперплоскостей, остаются в силе и для метода пирамид.

Другой отличительной особенностью метода пирамид является допущение непустоты пересечений множеств, образующих покрытие пространства итераций. Такие покрытия уже нельзя считать разбиениями. Множествами, образующими покрытия в методе пирамид являются геометрические фигуры, образованные гиперплоскостями, отделяющими вектора, от которых зависит выбранный результирующий вектор и вектора, от которых выбранный результирующий вектор не зависит. Этими множествами в случае двумерного пространства итераций будут треугольники, в случае трехмерного - пирамиды. Такой подход к формированию покрытия допускает многократное дублирование вычислений, связанных с одним вектором, входящим в разные множества одного покрытия. С одной стороны это дублирование нежелательно так как повышает время расчета по параллельному алгоритму, но с другой стороны оно позволяет производить вычисления над разделенной памятью не прибегая к синхронизациям и обмену сообщениями между задачами. По этому на рис. 1.12 отсутствуют каналы между задачами. Если время исполнения операторов тела цикла меньше времени передачи по сети (соединяющей области памяти, принадлежащие разным задачам) информации, от которой зависят операторы тела цикла, связанного с

результатирующим вектором, то разумнее продублировать вычисления, как это допускается в методе пирамид.

1.2.4 Метод параллелепипедов.

В методе параллелепипедов пространство итераций разбивается (в теоретико-множественном смысле) на параллелепипеды. За счет этого, в отличие от предыдущих методов, метод параллелепипедов позволяет параллелить простые циклы. Например, для программы

for i:=1 to 15 do x(i):=x(i-4);

пространство итераций будет одномерным (рис. 1.13), каждый вектор этого пространства будет зависеть от векторов, находящихся слева по оси i на расстоянии, кратном 4.

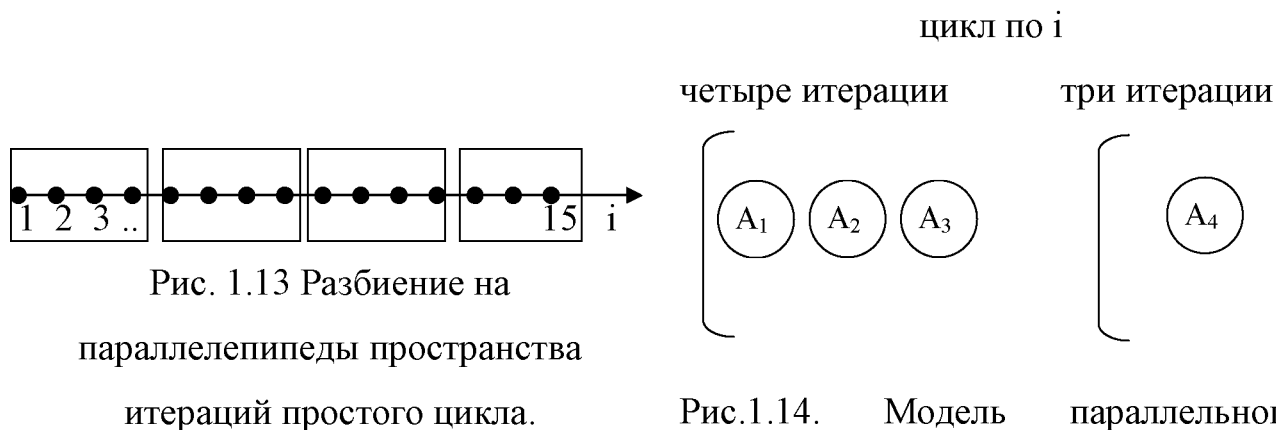


Рис. 1.13 Разбиение на параллелепипеды пространства итераций простого цикла.

Рис.1.14. Модель параллельного алгоритма для параллелепипедов в случае простого цикла.

Между множеством векторов в параллелепипеде, содержащем наибольшее количество векторов, и задачами алгоритма установим биекцию. Тогда задач будет 4 (рис. 1.14). Задаче под номером k ($k=1,4$) поручим следующие вычисления:

for i:=k to 15 step 4 do x(i):=x(i-4);

Первые три задачи произведут по 4 итерации цикла, четвертая 3. Алгоритм не нуждается в синхронизациях и передачах сообщений, так как операторы, содержащиеся в каждой задаче, соответствуют векторам, зависящим только от векторов данной задачи.

Аналогичный подход применяется для распараллеливания вложенной циклической конструкции общего вида. Пусть найдено разбиение пространства итераций на k -мерные параллелепипеды, при чем параллелепипед максимального объема характеризуется длиной сторон $p_1, p_2, \dots, p_k$ (предполагается существование меры в пространстве итераций). Тогда каждому вектору, находящемуся в таком параллелепипеде, поставим в соответствие одну задачу параллельного алгоритма. Такая задача будет характеризоваться вектором $g=(g_1+1, g_2+1, \dots, g_k+1)$, где g_q ($1 \leq q \leq k$) координата элемента пространства итераций, находящегося внутри параллелепипеда, в системе координат, оси которой совпадают с ребрами параллелепипеда. Для прямоугольного параллелепипеда общее число задач будет равно $V = \prod_{s=1}^k p_s$.

Если размерность параллелепипеда меньше размерности пространства итераций, то соответствующие $g_s=0, p_s=1$.

В параллельном алгоритме для задачи с вектором g установим следующую последовательность действий:

```

for i1:= g1+1 to n1 step p1 do
  for i2:= g2+1 to n2 step p2 do
    .....
    for ik:= gk+1 to nk step pk do
      T(i1,i2, ... ,ik);
      

```

Для общего случая в конце тела цикла необходима синхронизация между задачами, так как метод параллелепипедов не гарантирует независимость векторов итераций для любых двух задач.

Для примера распараллелим с помощью метода параллелепипеда циклическую конструкцию

```

for i:=1 to 7 do

```

```

  for j:=1 to 3 do
    x(i,j):=f(x(i-2,j-1));

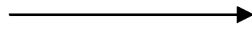
```

Для задачи с вектором g определим последовательность действий

```

for i:= g1+1 to 7 step p1 do
  for j:= g2+1 to 3 step p2 do
    x(i,j):=f(x(i-2,j-1));

```



и рассмотрим несколько разбиений пространства итераций на двумерные прямоугольные параллелепипеды (Рис. 1.15).

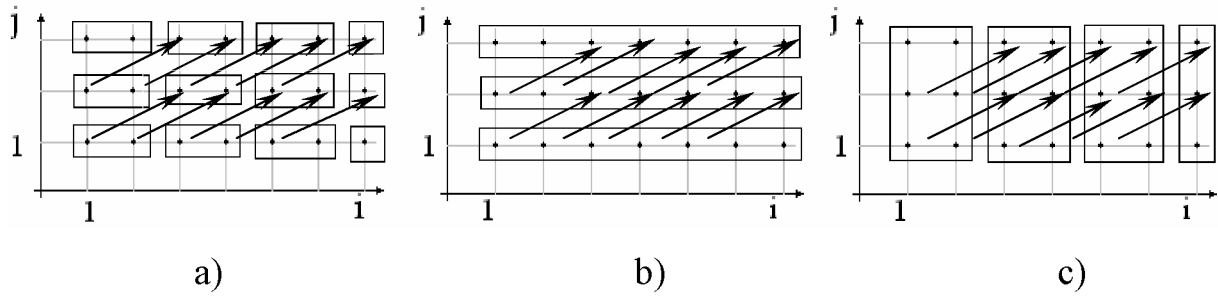


Рис. 1.15 Варианты разбиения пространства итераций на прямоугольные области, стрелками обозначены информационные зависимости.

Остановимся подробно на вычислениях по алгоритму, составленному для варианта а). Максимальный по объему прямоугольник содержит два вектора пространства итераций, следовательно, параллельный алгоритм будет состоять из двух задач. Для первой задачи $g^1=(1,1)$, для второй $g^2=(2,1)$. Прямоугольник характеризуется сторонами $p_1=2$ и $p_2=1$. В таблице 1.1 представлены параметры циклов, для которых будут произведены вычисления. Аналогичные таблицы можно составить для алгоритмов в случаях b) и c). Вариант b) характеризуется величинами $p_1=7$ и $p_2=1$, для задач $g^1=(1,1)$, $g^2=(2,1)$, $g^3=(3,1)$, $g^4=(4,1)$, $g^5=(5,1)$, $g^6=(6,1)$, $g^7=(7,1)$. Для варианта c) верно: $p_1=2$ и $p_2=3$, для задач $g^1=(1,1)$, $g^2=(2,1)$, $g^3=(1,2)$, $g^4=(2,2)$, $g^5=(1,3)$, $g^6=(2,3)$. Ускорения для вариантов а), b) и c) будут равны $21/12$, 7 и $21/4$. Очевидно вариант b) предпочтительнее остальных вариантов.

Отметим, что при разбиении а) нет необходимости в использовании синхронизаций, для разбиений b) и c) такая необходимость появляется.

Таблица 1.1

Параметры циклов задач параллельного алгоритма для варианта а)
разбиения пространства итераций (рис.1.15а)

Номер задачи	параметры циклов	значения параметров циклов при вычислениях по параллельному алгоритму			
I	i	1	3	5	7
	j	1, 2, 3	1, 2, 3	1, 2, 3	1, 2, 3
II	i	2	4	6	
	j	1, 2, 3	1, 2, 3	1, 2, 3	

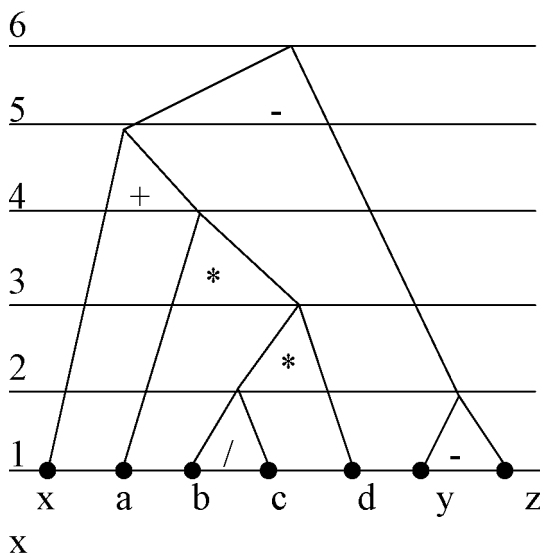
Как видно из примера, может существовать несколько разбиений пространства итераций (в общем случае конечное множество). Выбор наилучшего разбиения осуществляется методами целочисленного программирования для задачи максимизации целевой функции V при ограничениях, налагаемых на объем параллелепипеда зависимостями на пространстве итераций.

При больших значениях V не имеет смысла говорить об отображении векторов параллелепипеда с максимальным объемом, на множество задач алгоритма, как о биективной зависимости (пусть одной задаче допустимо соответствие нескольких таких векторов из пространства итераций). Иначе может сложиться ситуация, при которой количество задач многократно превысит число процессоров вычислительной системы. Например, при расчетах по простейшей явной разностной схеме для однородного двумерного линейного уравнения теплопроводности будем иметь дело с параллелепипедами, совпадающими с временными слоями сеточной области. Уже при дискретизации сетки по пространству 100×100 точек в параллелепипед войдет 10000 векторов. Не много найдется вычислительных систем, содержащих такое количество процессоров. Следует либо уменьшать объем параллелепипеда, что противоречит поставленной задаче целочисленного программирования, либо поручать одной задаче вычисления,

связанные с несколькими векторами пространства итераций, находящимися в таком параллелепипеде.

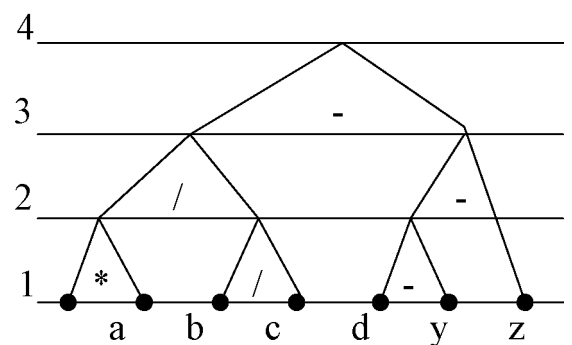
1.3 Распараллеливание выражений.

Большой интерес представляет проблема распараллеливания арифметических и логических выражений. Распараллеливаемый фрагмент последовательной программы принято представлять в удобной для исследования форме /2/. По ациклическим выражениям строится стандартный граф, по циклическим пространство итераций, для выражений записываются деревья. Так, выражение $t = (x + (a * ((b/c) * d))) - (y - z)$ может быть изображено, как показано на рис.1.16а.



a)

я
р
у
с
ы



b)

Рис. 1.16 Представление выражений в виде деревьев. а) - шести ярусное дерево для выражения t , б) - четырех ярусное дерево для выражения t'

В выражениях не всегда указан порядок применения операций. Даже тогда, когда в выражении не опущены скобки, имеется произвол в применении операций. Если же опускать некоторые скобки, то даже дерево выражения строится неоднозначно. Например, $a*b*c$ может быть "восстановлено" как $(a*b)*c$ и как $a*(b*c)$. Порядок вычисления может быть перераспределен или доопределен таким образом, что некоторые операции можно выполнять параллельно. Так, в выражении t можно параллельно

выполнить b/c и $y-z$. Если же мы перепишем t в форме $t'=((a*b)/(c/d))-((y-z)-x)$, то параллельно можно выполнить $a*b$, c/d и $y-z$. Дерево для t' приведено на рис.1.16.b

Если считать, что любая операция занимает одну единицу времени, то время вычисления t будет равно пяти единицам, а t' - трем, то есть благодаря параллельности исполнения время сокращается на две единицы. Вообще говоря, время реализации выражения при использовании всех резервов параллельности равно числу ярусов дерева минус единица. Это хорошо видно из рис. 1.17 Все операции, лежащие на одном ярусе дерева, можно выполнить параллельно.

Задача распараллеливания выражений ставится так. Построить алгоритм, который по каждому выражению дает эквивалентное ему с минимальной высотой дерева. При совмещении вычислений операций такое выражение реализуется за минимальное время. Эквивалентность понимается в смысле применения обычных законов ассоциативности, коммутативности и дистрибутивности. Соотношение дистрибутивности используется в редких случаях, так как неосмотрительное его применение может повлечь неприятности. Например, если в выражении $a*(b-c)$, a , b и c - большие положительные числа, то умножение $a*(b-c)$ происходит без коллизий, так как величины b и c "компенсируют" друг друга, в то время, как при порядке вычисления $a*b-a*c$ при умножениях может быть переполнение. Впрочем, подобное может случиться и при использовании других соотношений. Например, $(a*b)*c$ эквивалентно $a*(b*c)$, но при больших a и b и $c \ll 1$ при этом порядке умножения может возникнуть переполнение, которого бы не было при умножении $a*(b*c)$. Поэтому во многих случаях процедура распараллеливания распространяется только на бесскобочные фрагменты выражения, то есть на фрагменты, в которых очередность определяется лишь приоритетностью операций и никак не ограничена программистом.

Приведем алгоритм распараллеливания, применяющийся лишь для бесскобочных выражений. Используются операции $+$, $-$, $*$, $/$. Алгоритм

расписан достаточно детально в том смысле, что элементарными действиями его являются такие неделимые "машинные" операции, как сканирование, то есть считывание очередного символа выражения, засылка в память, сравнение двух элементов, проверка пустоты стека и приписывание символа X к выходной стороке Y , результат которого обозначается YX .

Для хранения текущей информации требуется следующая память:
 ячейки $v_i, i=1,2,3$, для хранения трех последовательных операндов;
 ячейки $z_i, i=1,2,3$, для хранения знаков операций после них;
 стеки z и v для хранения "запасного" операнда со знаком;
 стек e для выходного выражения;
 массив для записи промежуточных результатов $t(k)$;
 индексная переменная массива k .

Символ $\leftarrow$ обозначает засылку в ячейку или в стек, $scan(x_1, \dots, x_n)$ - сканирование n последующих символов входного арифметического выражения и засылку их по порядку в ячейки $x_1, \dots, x_n$. Предполагается, что сканирование начинается с первого операнда выражения. Выражение оканчивается пятью пробелами $_$. Первоначально $k=1$, стеки e, z, v - пустые.

Алгоритм.

Шаг 1. В стек e заносится знак при первом операнде выражения. Допускается, что он может быть минусом.

Шаг 2. $Scan(v_1, z_1, v_2, z_2, v_3, z_3)$. В шесть перечисленных ячеек заносятся шесть последовательных символов выражения.

Шаг 3. Если $(z_1=_)\wedge(z\neq\emptyset)$, то перейти на шаг 13. Если $(z_1=_)\wedge(z=\emptyset)$, то перейти на шаг 14. Если $(z_1\neq_)\wedge(z_2=_)$, то перейти на шаг 15.

Шаг 4. Если $(z=\emptyset)\wedge(e=-)\wedge(z_1=\pm)$, то $z\leftarrow\bar{}$ z . Если $(z=\emptyset)\wedge(e=-)\wedge(z_1=\pm)\wedge(z_2=\pm)$, то $z_1\leftarrow\bar{}$ z_1 . Этот шаг предназначен для изменения знака второго слагаемого на противоположный при предвидимом заключении соседних слагаемых в скобки, когда перед скобкой будет стоять минус. Если запасной стек не пуст, верхний элемент стека e - минус, а знак z_1 - плюс или минус, то $\bar{}$ z заносится на место z , то есть знак в запасном стеке z меняется на

противоположный. Противоположный знак для плюса - минус, для минуса - плюс, для умножения - деление, для деления - умножение. Аналогичное изменение производится в ячейке z_1 , если запасной стек пуст, е оканчивается минусом и в z_1, z_2 находятся аддитивные операции.

Шаг 5. Если $(z=\emptyset) \wedge (z_1 = \pm) \wedge (z_2 = */)$, то перейти на шаг 6. Если $(z \neq \emptyset) \wedge (z_1 = \pm)$, то перейти на шаг 7, иначе на шаг 9.

Шаг 6. $v \leftarrow v_1$; $z \leftarrow e$; $e \leftarrow z_1$. Операнд из v_1 и знак, завершающий е засылаются в "запасные" стеки. В е засылается знак из z_1 . Переход на шаг 8.

Шаг 7. $t(k) \leftarrow (v_1 z v)$; $e \leftarrow t(k) z_1$; $k \leftarrow k+1$. Пара операндов v_1 и v связываются знаком z , заключаются в скобки и присваиваются промежуточному результату $t(k)$. Последний приписывается к е, вслед выписывается знак z_1 . Значение счетчика k увеличивается на единицу.

Шаг 8. $v_1 \leftarrow v_2$; $z_1 \leftarrow z_2$; $v_2 \leftarrow v_3$; $z_2 \leftarrow z_3$; $\text{scan}(v_3, z_3)$. Рассматриваемый фрагмент выражения перемещается на две позиции влево. В освободившиеся ячейки v_3, z_3 считываются два последующих символа выражения. Переход на шаг 3.

Шаг 9. $t(k) \leftarrow (v_1 z_1 v_2)$; $e \leftarrow t(k) z_2$; $k \leftarrow k+1$. Формируется промежуточный результат $t(k)$, который приписывается к выражению е, вслед выписывается знак z_2 . Значение счетчика увеличивается на единицу.

Шаг 10. Если $(z_1 = */) \wedge (z_2 = */) \wedge ((z_3 = \pm) \vee (z_3 = _))$, то перейти на шаг 11, иначе на шаг 12.

Шаг 11. $e \leftarrow v_3 z_3$; Если $z_3 = \pm$, то перейти на шаг 2, иначе конец разбора.

Если знаками умножения и деления связаны три соседних операнда, а далее следует другой знак или пробел, то третий сомножитель и следующий за ним знак просто приписываются к формируемому выражению.

Шаг 12. $v \leftarrow v_3$. Если $(z_2 = /) \wedge (z_3 = */)$, то $z_1 \leftarrow \lceil z_3$, иначе $z_1 \leftarrow z_2$. $\text{Scan}(v_2, z_2, v_3, z_3)$. Переход на шаг 3.

Здесь рассматриваемый фрагмент выражения передвигается влево на четыре позиции, но если перед парой следующих операндов стоит операция

деления, а операция между ними - мультипликативная, то она меняется на обратную. Далее сканируются четыре символа из входного выражения и делается переход к началу разбора.

Шаг 13. Если $e = -$, то $z \leftarrow \lceil z$; $t(k) \leftarrow (v_1 z v)$; $e \leftarrow e t(k)$. Конец разбора.

Шаг 14. $e \leftarrow v_1$. Конец разбора.

Шаг. 15. Если $(e = -) \wedge (z_1 = \pm)$, то $z_1 \leftarrow \lceil z_1$; $t(k) \leftarrow (v_1 z_1 v_2)$; $e \leftarrow t(k) z v$. Конец разбора.

Последние три шага служат для завершения разбора. На шаге 13 в промежуточный результат "свертываются" "запасной" и последний оставшийся операнды. На шаге 14 в выражению просто приписывается оставшийся операнд. На шаге 15 промежуточный результат формируется из двух оставшихся операндов, затем приписывается стек запасного операнда, который может быть и пустым.

Приведенный алгоритм описывает один из проходов, после которого некоторые из подвыражений сводятся в промежуточные результаты $t(k)$. После этого алгоритмы повторяются до тех пор, пока все выражение не сведется к единственному $t(k)$. Ниже показано преобразование выражения $x + a * b / c * d - y + z$, полученного из t раскрытием скобок, после серии последовательных проходов.

1. Входная строка $+x + a * b / c * d - y + z$ _____.
2. Результат 1-го прохода: $+t(1)/t(2)-t(3)+x$, где $t(1)=(a*b)$, $t(2)=(c/d)$, $t(3)=y-z$.
3. Результат 2-го прохода: $+t(4)-t(5)$, где $t(4)=(t(1)/t(2))$, $t(5)=(t(3)-x)$.
4. Результат 3-го прохода: $+t(6)$, где $t(6)=(t(4)-t(5))$.
5. Выходная строка $((a*b)/(c/d))-((y-z)-x)$. Результирующее выражение совпало с t' . Алгоритм максимально распараллеливает входное выражение, то есть на выходе дает выражение с деревом минимальной высоты.

Любое арифметическое выражение без унарного минуса с n операндами может быть вычислено на $n/2$ процессорах за время порядка $\log_2(n)$, при этом сложность преобразования в максимально параллельную

форму равна $O(n)$. Этот результат выглядит сильным с теоретической точки зрения, но, к сожалению, на практике редко встречаются выражения с большим числом операндов, чтобы могла сыграть существенную роль подобная логарифмическая оценка. Хотя в ряде случаев методы распараллеливания выражений эффективно применимы к вычислениям типа $\sum_{i=1}^m \prod_{j=1}^n x_{ij}$, где при больших n и m уже сказывается определяемое ими

логарифмическое ускорение.

В заключении главы остановимся на следующем поучительном примере. Может показаться, что овладев методами распараллеливания последовательных программ изучение курса можно завершить. По последовательному алгоритму записывается последовательная программа. В результате ее распараллеливания, с помощью методов настоящей главы, синтезируется параллельный алгоритм и кодируется параллельная программа.

Пусть дана система рекуррентных соотношений $x_i = a_i x_{i-1} + b_i$, $x_0 = b_0$. Последовательные вычисления по данной системе для нахождения x_n программируются простым циклом.

```
x:=b[0];
for i:=1 to n do x:=a[i]*x+b[i];
```

Пространство итераций, связанное с этим циклом и зависимости (информационные и конкуренционные зависимости в данном случае совпадают) показаны на рис. 1.17.

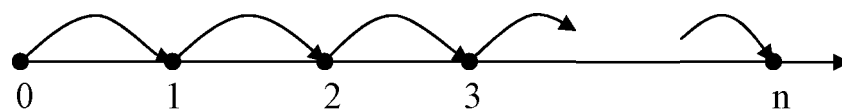


Рис. 1.17. Пространство итераций и зависимости для циклической конструкции, для которой невозможно распараллеливание.

Каждый вектор пространства итераций зависит от предыдущего, по этому вычисления, организованные по любым двум векторам не могут происходить одновременно, общее число операций оценивается как $O(n)$.

Оставим рассмотрение последовательной программы, обратившись к последовательному алгоритму. Запишем систему рекуррентных соотношений в матричном виде, приняв $X_i = \begin{pmatrix} x_i \\ 1 \end{pmatrix}$.

$$\begin{aligned} X_n &= \begin{pmatrix} a_n & b_n \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{n-1} \\ 1 \end{pmatrix} = \begin{pmatrix} a_n & b_n \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_{n-1} & b_{n-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{n-2} \\ 1 \end{pmatrix} = \\ &= \begin{pmatrix} a_n & b_n \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_{n-1} & b_{n-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_{n-2} & b_{n-2} \\ 0 & 1 \end{pmatrix} \times \dots \times \begin{pmatrix} x_0 \\ 1 \end{pmatrix} = \prod_{i=n}^1 A_i X_0 \end{aligned}$$

Перемножение n -матриц можно производить параллельно (подробности в Главе 3), при этом количество операций оценивается функцией $O(\log_2 n)$. Следовательно, распараллеливая алгоритм, а не программу, можно добиться лучших результатов.

ГЛАВА 2. СИНТЕЗ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ

2.1. Понятие параллельного алгоритма.

Под алгоритмом будем понимать точное предписание, определяющее вычислительный процесс, идущий от варьируемых начальных данных к исходному результату (Гост - 19.781 - 74 - определение алгоритма). Под исходным подразумевается окончательный результат. Соответственно, параллельный алгоритм определяет параллельный вычислительный процесс, в котором хотя бы два действия выполняются одновременно.

Свойства алгоритма.

1. *Однозначность.* Алгоритм не должен допускать двух разных трактований.
2. *Результативность.* Для нахождения результата (или убежденности в его отсутствии) требуется исполнение конечного числа шагов алгоритма.
3. *Массовость.* Задачи, отличающиеся лишь начальными данными, могут решаться с помощью одного алгоритма.
4. *Дискретность.* Алгоритм должен допускать разбиение на части (шаги алгоритма).

Принято четыре основных формы представления алгоритмов.

1. *Графическая форма.* Последовательные алгоритмы представляются, как правило, в виде блок-схем. При графической записи параллельных алгоритмов также используют символы, определенные в модели канал/задача (параграф 1.2).
2. *Словесное описание.* Если по словесному описанию можно заключить, что алгоритм удовлетворяет четырем перечисленным свойствам и подходит под определение, то оно может считаться формой представления алгоритма.
3. *Запись в виде пошаговых инструкций.* По свойству дискретности любой алгоритм должен допускать разбиение на шаги. Действия, производимые при исполнении любого шага описываются отдельно от действий других шагов.
4. *Представление в виде текста программы на алгоритмическом языке.* Однако не всякая программа, написанная на таком языке, будет алгоритмом.

Например, программа, допускающая "зацикливание" вычислительного процесса, не удовлетворяет свойству результативности.

Распространенной ошибкой является непонимание отличия алгоритма от вычислительного процесса, им порождаемого. Алгоритм, даже оформленный в виде программы - это идея (в смысле Платона), а вычислительный процесс - действия, происходящие в вычислительной системе (вполне материальной). Таким образом, алгоритм и вычислительный процесс располагаются в разных мирах (реальном и воображаемом), что обуславливает различие в подходах к их изучению. Сначала появляется идея (алгоритм), потом идея порождает действия (вычислительный процесс). Если алгоритм принято синтезировать, а затем анализировать, то с вычислительным процессом поступают наоборот. Моделирование вычислительного процесса производят исключительно для его анализа, а анализ алгоритма служит для задач синтеза. Поэтому, изучая вычислительные процессы, мы уделяли основное внимание анализу, говоря об алгоритмах, обратимся к синтезу.

Приступая к синтезу параллельного алгоритма по известному последовательному, полезно оценить принципиальную возможность эффективного распараллеливания. Например, производя вычисления по разностным уравнениям явной схемы $L_h U_h = f_h$ на трехмерной сеточной области D_h , разобьем область на две части D_1 и D_2 , организовав обмен значениями сеточной функции U_h на границе разбиения Γ (рис. 2.1). В подобластях D_1 и D_2 вычисление сеточной функции U_h можно производить одновременно (параллельно), учитывая значения U_h на границе соседней подобласти.

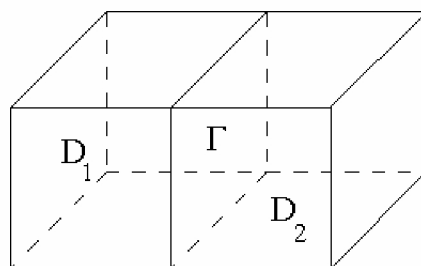


Рис. 2.1 Разбиение сеточной области D_h на две подобласти D_1 и D_2 , разделенные границей Γ .

Пусть для некоторой вычислительной системы время вычислений по области D_h составит T_D . За $T_{D1}=T_{D2}=\frac{1}{2}T_D$ примем время вычислений в подобластях D_1 и D_2 . Если бы затраты времени ограничивались длительностью арифметических операций, то данная задача в параллельном варианте решалась бы вдвое быстрее, чем в последовательном. К сожалению, к этим затратам следует добавить время на передачу и прием значений функции в граничных узлах сетки. Тогда $T_{D1}=T_{D2}=\frac{1}{2}T_D+T_{\text{пересылок}}$. При $T_{\text{пересылок}}>\frac{1}{2}T_D$, что весьма возможно, $T_{D1}>\frac{1}{2}T_D$ и $T_{D2}>\frac{1}{2}T_D$, тогда время вычислений по параллельному алгоритму превзойдет время вычислений по последовательному. В синтезе подобных алгоритмов нет необходимости.

Пока существует возможность производства вычислений с приемлемыми затратами времени по последовательному алгоритму, в параллельном алгоритме также нет необходимости.

2.2. Восходящий подход к синтезу параллельных алгоритмов.

Различают два подхода к синтезу параллельных алгоритмов. В восходящем подходе объект исследования (последовательный алгоритм или численный метод) сразу дробится на мелкие части, которые в процессе построения параллельного алгоритма объединяются во все более крупные до тех пор, пока характеристики алгоритма не удовлетворят исследователя. Нисходящий подход подразумевает разбиение исходной задачи на крупные блоки, которые в дальнейшем дробятся на более мелкие.

Этапы синтеза параллельного алгоритма восходящим методом представлены на рис. 2.2. /3/

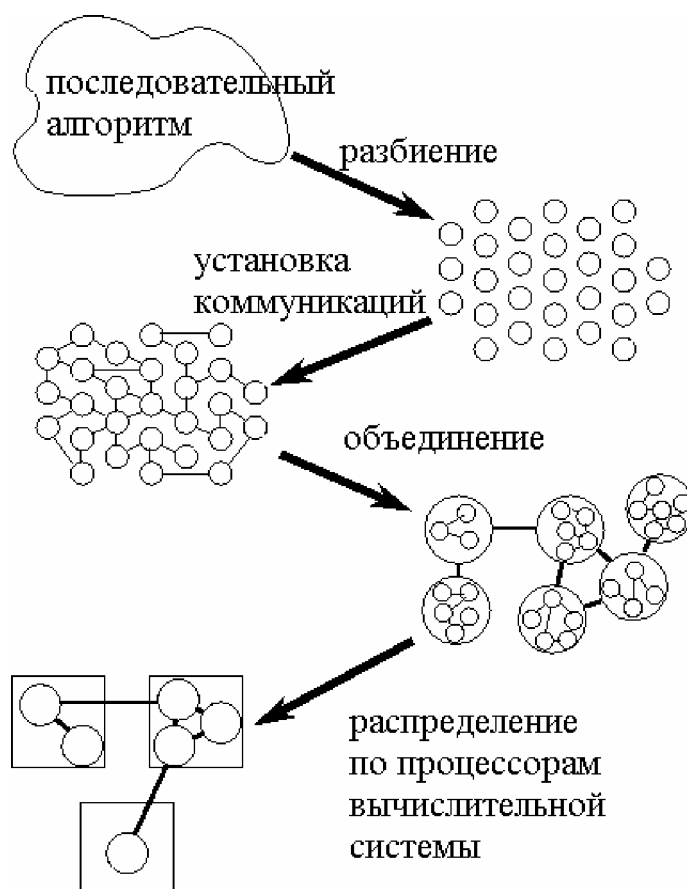


Рис. 2.2. Этапы синтеза параллельного алгоритма восходящим методом. Остановимся подробно на трех из них.

2.2.1. Разбиение.

Цель разбиения - формирование как можно большего числа задач параллельного алгоритма. Чем больше задач появится при разбиении, тем меньше ограничений стеснит фантазию разработчика на этапе сборки. Вместе с тем, не стоит обращаться к крайностям, приписывая каждой задаче, формируемой на данном этапе, одну арифметическую операцию (далее разбивать некуда). Следует выделять в одну задачу наименьшую, но характерную именно для данного последовательного алгоритма, часть. Если дальнейшее дробление этой части приведет к тому, что она станет присущей более широкому классу алгоритмов, то такое дробление производить не следует.

В качестве примера рассмотрим решение сеточных уравнений

$$\frac{U_i^{k+1} - U_i^k}{h_t} = \frac{U_{i-1}^k - 2U_i^k + U_{i+1}^k}{h_x^2}$$

простейшей явной разностной схемы для однородного одномерного линейного нестационарного уравнения теплопроводности на сетке $D_h = \{(t_k, x_i), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=0, 1, \dots, I=L/h_x\}$. Произведя максимально мелкое разбиение D_h по направлению x , получим I задач (рис. 2.3).

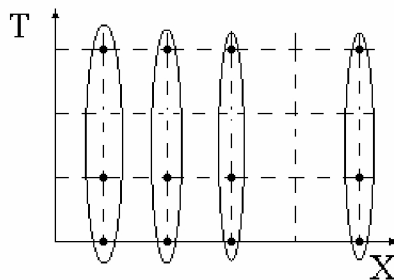


Рис. 2.3 Разбиение на задачи сеточной области D_h . Обведены узлы сетки, принадлежащие одной задаче.

Каждая задача решает одно разностное уравнение при фиксированном k . Увеличивая количество задач на одном временном слое, разобьем вычисления по одному разностному уравнению на несколько частей. Получим набор арифметических операций, по которым уже нельзя понять, что за алгоритм мы распараллеливаем.

Другое требование к образованным в результате разбиения задачам параллельного алгоритма - возможность их одновременного исполнения. В предыдущем примере нельзя увеличить число задач за счет разбиения по оси T , так как две задачи, характеризующиеся одним индексом i , не смогут исполняться одновременно.

На рис. 2.1. и 2.3. разбивалась область данных, с которыми работает последовательный алгоритм. В зависимости от этого разбиения распределялись вычисления между задачами параллельного алгоритма. Такой подход получил название декомпозиции данных. При функциональной декомпозиции сначала разбиваются вычисления, потом между задачами

распределяются данные, с этими вычислениями связанные. Различия двух подходов к разбиению последовательного алгоритма на задачи иллюстрируются следующим примером.

Пусть на сеточной области $D_h = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=0, 1, \dots, I=L_x/h_x, y_j = jh_y, j=0, 1, \dots, J=L_y/h_y\}$ записана система явных разностных уравнений

$$\begin{aligned} L_h^1 U_h &= f_h^1(V_h) \\ L_h^2 V_h &= f_h^2(U_h) \end{aligned} \quad (2.1).$$

Для простоты ограничимся разбиением на две задачи параллельного алгоритма. Производя декомпозицию данных разделим D_h на $D_1 = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=0, 1, \dots, I_1=L_x/(2h_x), y_j = jh_y, j=0, 1, \dots, J=L_y/h_y\}$ и $D_2 = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=I_1+1, I_1+2, \dots, L_x/h_x, y_j = jh_y, j=0, 1, \dots, J=L_y/h_y\}$. Тогда первая и вторая задачи решают оба уравнения системы (3.1) для разных подобластей, обмениваясь значениями сеточных функций U_h^Γ и V_h^Γ на границе (рис. 2.4 а).

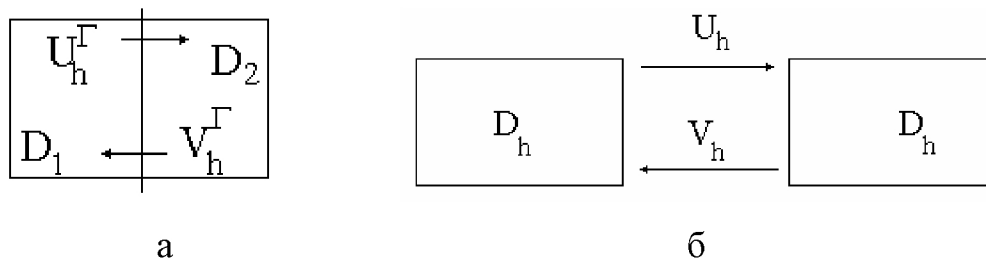


Рис. 2.4 Разбиение сеточной области и организация коммуникаций между задачами при а) декомпозиции данных, б) при функциональной декомпозиции.

При функциональной декомпозиции в первую очередь распределяются уравнения системы. Вычисления по первому уравнению включим в первую задачу, по второму - во вторую. На новом временном слое первая задача вычислит значение U_h , вторая задача значение V_h на всей области D_h . При этом каждой задаче придется хранить обе сеточные функции на всей области, обмениваясь вычисленными значениями (рис. 2.4 б). Иначе невозможно продолжить вычислений по схеме (2.1), правая часть первого уравнения

которой зависит от V_h (следовательно, первая задача вынуждена хранить значения этой функции на D_h), второго уравнения от U_h (вторая задача хранит U_h на D_h).

Производя такое же количество арифметических операций одна задача алгоритма, полученного посредством функциональной декомпозиции, занимает вдвое больше памяти и передает/получает в I раз больше данных, чем одна задача алгоритма, записанного с помощью декомпозиции данных. Очевидно, декомпозиция данных, для этого примера, предпочтительней функциональной декомпозиции. При функциональной декомпозиции наилучшим образом разбиваются вычисления (так как это разбиение происходит в первую очередь), распределение данных происходит менее удачно (оно целиком зависит от разбиения вычислений и носит подчиненный характер). Декомпозиция данных наоборот, предусматривает оптимальное распределение данных между задачами и менее удачное распределение вычислений. Между ветвями вычислительного процесса, порожденного параллельным алгоритмом, происходит обмен данными, а не вычислений (при необходимости вычисления дублируются, а не пересылаются), поэтому оптимально распределить данные между задачами важнее, чем красиво разбить уравнения.

Зададимся целью разбить исследуемый последовательный алгоритм на максимально большое число задач L , положив для простоты $J=1$. Декомпозиция данных позволяет синтезировать I - задачный параллельный алгоритм (рис. 2.3). Осуществляя функциональную декомпозицию, при $L>2$ вступим в противоречие с требованием разбиения на характерные для данного алгоритма части, ибо распределять между задачами придется члены разностных уравнений (рис. 2.5).

$$\begin{array}{c}
 L_h^1 \left\{ U_h = f_h^1(V_n) \right. \\
 L_h^2 \left\{ V_h = f_h^2(U_n) \right. \\
 1 \quad 2 \quad \dots \quad L
 \end{array}$$

задачи

Рис. 2.5. Функциональная декомпозиция системы разностных уравнений (2.1) для произвольного числа задач параллельного алгоритма.

При измельчении сетки D_h (для повышения точности разностного решения) использование декомпозиции данных позволит увеличить количество задач параллельного алгоритма, а применение функциональной декомпозиции ограничит исследователя фиксированным числом задач при любой сеточной области.

Определим свойство масштабируемости параллельного алгоритма, как возможность увеличивать число задач с возрастанием объема данных. Немасштабируемый алгоритм подобен последовательному, длительность вычислений по нему возрастает с увеличением области данных. Масштабируемый алгоритм позволяет компенсировать изменение объема данных таким изменением количества задач, при котором длительность вычислений не увеличивается. Декомпозиция данных, в отличие от функциональной декомпозиции, характеризуется масштабируемостью синтезируемых алгоритмов.

Несмотря на изложенные преимущества, не следует отдавать безусловного предпочтения декомпозиции данных. После применения функциональной декомпозиции нет необходимости в доказательстве сходимости и устойчивости используемого численного метода - он остался прежним. Проводя декомпозицию данных, приходится приспособливать вычисления к разбиению данных, зачастую модифицируя численный метод, который уже может не обладать сходимостью и устойчивостью. Разработке и обоснованию новых вычислительных методов иногда удобнее предпочесть синтез алгоритма, основанного на функциональной декомпозиции.

Кратко повторим основные требования к этапу разбиения.

1). Последовательный алгоритм следует разбивать на максимально большое число задач, каждая задача параллельного алгоритма содержит минимальный, но характерный участок последовательного.

2). Задачи должны быть сбалансированными - содержать одинаковый объем вычислений и данных.

3). Предпочтение отдается масштабируемым параллельным алгоритмам.

2.2.2. Установка коммуникаций.

Будем говорить, что задача I_1 , данного параллельного алгоритма, зависит от задачи I_2 , того же алгоритма, если найдется хотя бы одно действие d_1 параллельной ветви, порожденной задачей I_1 , которое в любом вычислительном процессе, определенном данным параллельным алгоритмом, всегда будет следовать за некоторым действием d_2 параллельной ветви, порожденной задачей I_2 .

На этапе установки коммуникаций зависимые задачи связываются каналами, по которым передаются данные. Под коммуникациями параллельного алгоритма будем понимать совокупность каналов и их топологию. Различают следующие типы коммуникаций.

1). Локальные и глобальные коммуникаций.

Если хотя бы одна задача параллельного алгоритма связана каналами со всеми остальными задачами, то уместно говорить об алгоритме с глобальными коммуникациями. В алгоритме с локальными коммуникациями любая задача связана с "небольшим" числом других задач. Эти задачи назовем соседями данной задачи, а "небольшим", для определенности, будем считать количество соседей, не превышающее размерность гиперкуба, вершины которого соответствуют задачам исследуемого алгоритма. Если число задач алгоритма не составляет степени двойки, используем для оценки локальности ближайшую большую степень двойки.

Построим параллельный алгоритм решения разностных уравнений

$$\frac{U_{i,j}^{k+1} - U_{i,j}^k}{h_t} = \frac{U_{i-1,j}^k - 2U_{i,j}^k + U_{i+1,j}^k}{h_x^2} + \frac{U_{i,j-1}^k - 2U_{i,j}^k + U_{i,j+1}^k}{h_y^2} \quad (2.2)$$

явной схемы для однородного двумерного линейного нестационарного уравнения теплопроводности на сетке $D_h = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=1, 2, 3, y_j = jh_y, j=1, 2, 3\}$. Произведя на этапе разбиения декомпозицию данных, поставим в соответствие каждому узлу сеточной области одну задачу алгоритма (рис. 2.6).

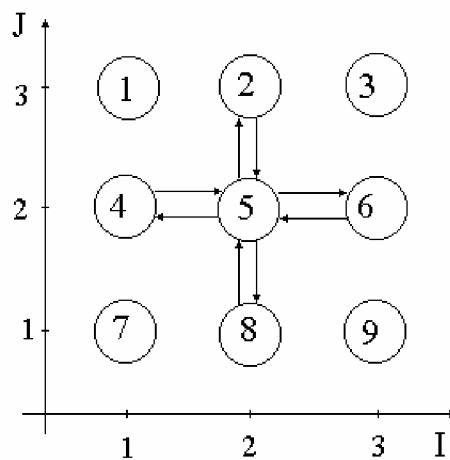


Рис. 2.6. Коммуникации 5-ой задачи в параллельном алгоритме решения разностных уравнений (2.2).

В силу особенностей шаблона для (2.2) задачи с номерами 1,3,7,9 связываются каналами с 2 соседями, задачи 2,4,6,8 с 3 соседями, задача 5 с 4 соседями (рис. 2.6). Например, задача 5, для решения уравнения (2.2) относительно $U_{2,2}^{k+1}$ нуждается в получении значений: $U_{2,3}^k$ от задачи 2, $U_{3,2}^k$ от задачи 6, $U_{2,1}^k$ от задачи 8 и $U_{1,2}^k$ от задачи 4. Перечисленные задачи, в свою очередь, принимают значение $U_{2,2}^k$ от 5-ой задачи. Составленный алгоритм характеризуется локальными коммуникациями - максимальное количество соседей (4 у 5-ой задачи) не превышает установленного предела.

Установим коммуникации параллельного алгоритма взятия определенного интеграла по восьми значениям подынтегральной функции.

Восемь задач, хранящих значения известной сеточной функции в различных узлах, производят вычисления по квадратурной формуле прямоугольников, находя восемь интегральных сумм. Необходимо, связав каналами задачи параллельного алгоритма, сложить полученные числа.

Пусть задачи 2,3,4,5,6,7,8 передадут найденные интегральные суммы 1-ой задаче, которая завершит вычисления по алгоритму (рис. 2.7). Коммуникации, организованные таким образом, глобальны. Рассмотрим присущие им недостатки.

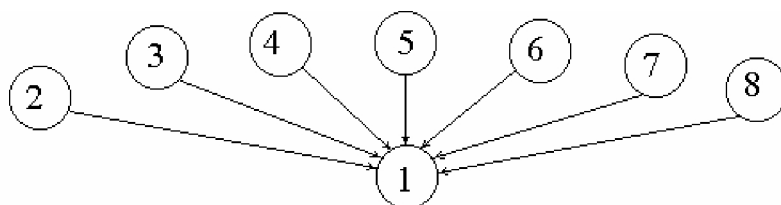


Рис. 2.7. Топология глобальных коммуникаций параллельного алгоритма взятия определенного интеграла по восьми значениям подынтегральной функции.

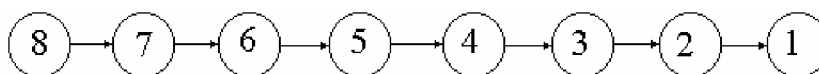
Первая задача, как и всякая другая задача параллельного алгоритма, порождает последовательный вычислительный процесс, в котором все действия, в том числе принятие по каналам интегральных сумм и их сложение, происходят последовательно. Если для расчета по квадратурной формуле прямоугольников требуется одна арифметическая операция (ее одновременно произведут все задачи алгоритма), то для сложения интегральных сумм - семь. Таким образом, произведя одну арифметическую операцию, задачи 2,3,4,5,6,7 в дальнейших вычислениях не участвуют, а задача 1 производит еще семь сложений. Для сокращения времени вычислений следует равномернее распределять операции между задачами алгоритма.

Более того, при реализации такого алгоритма на гиперкубе (наиболее распространенной кластерной конфигурации) две задачи из семи не смогут передать вычисленные значения интегральных сумм непосредственно первой

задаче, между ними и первой задачей отсутствует прямая связь по сети, связывающей ЭВМ вычислительной системы.

Алгоритмы с локальными коммуникациями свободны от перечисленных недостатков. Например, при сгущении сетки и увеличении количества задач на рис. 2.6, число соседей для любой задачи алгоритма не превысит четырех. Следовательно, каждая задача произведет не более четырех последовательных приемов/передач по сети гиперкуба и решит свое сеточное уравнение одновременно с остальными задачами алгоритма.

На рис. 2.8 предложена схема локальных коммуникаций алгоритма численного интегрирования, которая позволяет реализовывать вычисления на гиперкубе, причем арифметические операции распределены равномерно между задачам. Задача 7 получая интегральную сумму от 8-ой задачи, производит суммирование и передает результат задаче 6. Задача 6 получает интегральную сумму от 7-ой задачи, производит суммирование, передает результат задаче 5, и так далее до задачи 1.



2.8. Топология локальных коммуникаций параллельного алгоритма взятия определенного интеграла по восьми значениям подынтегральной функции.

К сожалению, при сложении интегральных сумм всегда простаивают, не производя вычислений, семь задач из восьми. Например, когда сложение производит задача 5, задачи 8,7,6 уже передали результаты своих вычислений и в дальнейшей работе алгоритма не участвуют, а задачи 4,3,2,1 еще не получили необходимых данных для начала суммирования. Хотя нам удалось адаптировать алгоритм к сетевой конфигурации гиперкуба, суммирование все еще производится последовательно.

В таких случаях принято использовать правило "разделяй и властвуй". Его суть заключается в переходе от линейной структуры коммуникаций (рис. 2.8) к коммуникациям на дереве (рис.2.9).

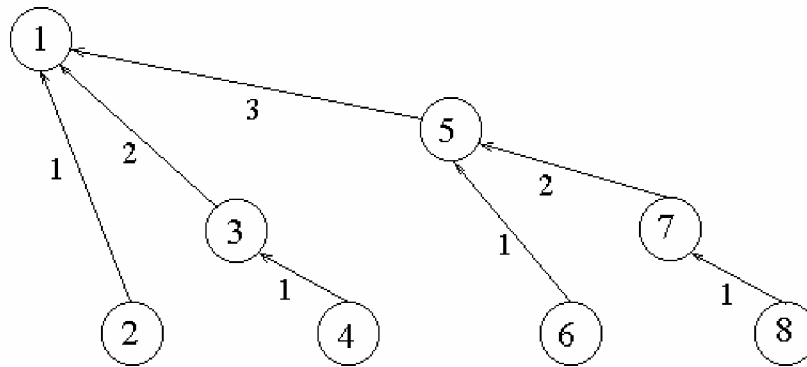


Рис. 2.9. Организация коммуникаций по правилу "разделяй и властвуй".

Под каналами цифрами обозначена очередность передачи данных.

На первом шаге алгоритма все задачи вычисляют интегральные суммы. На втором шаге задачи 2,4,6,8 передают задачам 1,3,5,7 найденные значения (рис. 2.9), которые прибавляются к результатам, вычисленным задачами - получателями. Третий шаг алгоритма - передача данных от задач 3,7 к задачам 1,5 и аналогичное суммирование. На четвертом, последнем шаге алгоритма, задача 1 получая от задачи 5 необходимую информацию находит значение определенного интеграла.

Максимальное количество соседей (три у 3-ей задачи) не превышает размерность гиперкуба с восьмью вершинами - коммуникации предложенного алгоритма локальны. Более того, такая структура коммуникаций позволяет на втором шаге алгоритма произвести одновременно 4 сложения и 2 сложения на третьем шаге. Для аналогичного алгоритма, состоящего из L задач, сложение интегральных сумм потребует не $L-1$ последовательное применение операции суммирования, а $\log_2 L$ таких операций.

2). Синхронные и асинхронные коммуникации различаются по аналогии с синхронным и асинхронным обменом сообщениями между параллельными ветвями вычислительного процесса. К преимуществу асинхронных коммуникаций следует отнести возможность продолжения вычислений, если ожидаемое по каналу сообщение еще не поступило. Это полезно при вычислениях по несбалансированному алгоритму, синхронные

коммуникации в этом случае увеличат время расчета. Использование асинхронных коммуникаций в хорошо сбалансированном алгоритме излишне.

3). Статические и динамические коммуникации.

Если на множестве задач параллельного алгоритма, при неизвестных начальных данных, можно установить все отношения зависимости, то такой алгоритм характеризуется статическими коммуникациями. Для него топология каналов не меняется при варьировании начальных данных. Если все отношения зависимости до получения начальных данных однозначно установить нельзя - говорим об алгоритме с динамическими коммуникациями. Топология каналов такого алгоритма может меняться при различных входных данных. Наличие в алгоритме динамических коммуникаций серьезно осложняет его анализ. Перед реализацией вычислений по такому алгоритму следует с осторожностью выбирать сетевую конфигурацию вычислительной системы. Оптимальная при одних начальных данных конфигурация может оказаться неудачной при других.

4). Регулярные и нерегулярные коммуникации.

Предположим, что все задачи исследуемого параллельного алгоритма, обозначив точками, удалось разместить в пространстве R^n таким образом, что задачи, зависящие от любой заданной, составляют правильную геометрическую фигуру (n угольник, тор, сферу и т.д.), дерево или другую осмысленную структуру. Если это утверждение верно и для всех задач, от которых зависит любая заданная задача, то коммуникации такого алгоритма можно назвать регулярными. Когда получившиеся выше топологии нельзя описать аналитическими выражениями, говорят о нерегулярных коммуникациях параллельного алгоритма. Для реализации вычислений по алгоритму с нерегулярными коммуникациями сложно подобрать

вычислительную систему с простой сетевой конфигурацией (гиперкуб, тор, и т.п.).

Кратко остановимся на способе передачи группы данных по одному каналу. К особенностям передачи сообщений по сети вычислительной системы относятся: необходимость создания почтовых ящиков и пакетный способ передачи. Ко времени передачи данных следует прибавить время, затрачиваемое на организацию этой передачи: создание почтового ящика, управление его работой и удаление ящика. Пакетный способ передачи заключается в посылке сообщения не сразу целиком, а по частям, пакетами. Если объем одного пакета превышает объем передаваемых данных, то время коммуникации будет определяться размером пакета, а не данных. С учетом этого, зависимость времени передачи сообщения от его объема будет не линейной функцией, а ступенчатой (рис.2.10).

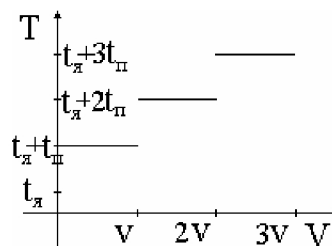


Рис. 2.10 Зависимость времени передачи сообщения T от его объема V .

v -объем одного пакета, $t_я$ - время организации передачи (создание, работа и удаление почтового ящика), $t_п$ - время передачи одного пакета по сети.

Представим ситуацию, когда по одному каналу необходимо передать два массива данных одинакового объема $v/2$. При раздельной передаче (каждый массив передается в виде отдельного сообщения) время пересылки составит $2t_я+2t_п$ ($t_я+t_п$ для каждого сообщения). Если поместить два массива в один, объемом v , и передать одним сообщением, то время пересылки сократится вдвое (при том, что в обоих случаях пересылается одинаковый объем данных). Таким образом, следует отдавать предпочтение передаче

больших объемов данных в меньшем количестве сообщений над передачей меньших объемов в большем количестве.

Сформулируем основные рекомендации к этапу установки коммуникаций.

1). Желательно избавляться от глобальных коммуникаций, переводя их в локальные по правилу "разделяй и властвуй".

2). Задачи хорошо сбалансированного алгоритма связываются одинаковым количеством каналов.

3). Использование регулярных статических коммуникаций упрощает организацию вычислительного процесса (на большинстве вычислительных систем) по созданному алгоритму.

2.2.3. Объединение.

Вернемся к уравнениям (2.2), рассматривая их на сетке $D_h = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=1, 2, \dots, 8; y_j = jh_y, j=1, 2, \dots, 8\}$. После этапов разбиения и установки коммуникаций получим 64 задачи параллельного алгоритма. Вычисляя $U_{i,j}^{k+1}$, отмеченная на рис. 2.11а задача произведет 13 арифметических операций, на которые придется 4 приема значений $U_{i-1,j}^k$, $U_{i+1,j}^k$, $U_{i,j-1}^k$, $U_{i,j+1}^k$ и 4 передачи $U_{i,j}^k$ соседним задачам. Все значения передаются в разных пакетах, так как предназначены разным задачам и пойдут по разным каналам. На одну операцию посылки или приема пакета придется 1,625 арифметических операций. Предположим, что данный алгоритм реализован на вычислительной системе, процессоры которой производят 100 арифметических операций в секунду, а данные по сети передаются со скоростью 1 пакет в секунду. Тогда 1,6% от времени вычислений будет связано с производством арифметических операций, а 98,4% с передачей данных по сети. Общее время вычислений на одном слое

по k составит 8,13 с. Вычисления по последовательному алгоритму, на данной вычислительной системе, потребуют 8,32 с.

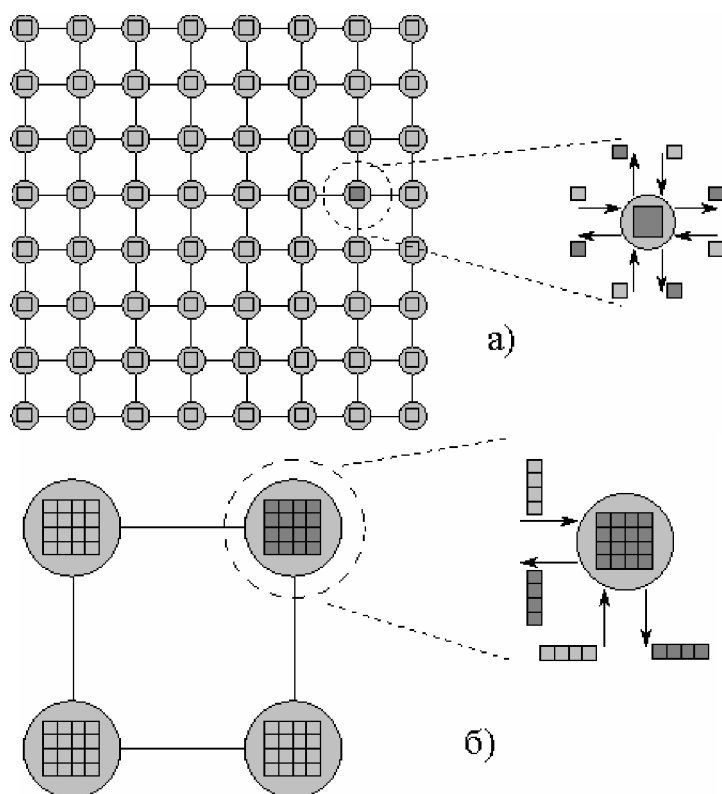


Рис. 2.11 Объединение задач при построении параллельного алгоритма решения разностных уравнений (2.2). а) - алгоритм до объединения, б) - после.

Объединим по 16 задач параллельного алгоритма в одну, как это показано на рис. 2.11б, получив четырехзадачный алгоритм. Для него каждая задача производит 208 арифметических операций, связанных с пересылкой 4 пакетов (в каждом по 4 значения сеточной функции). Для рассматриваемой вычислительной системы время работы по алгоритму составит 6,08 с. Объединение задач алгоритма позволило на четверть сократить время вычислений по нему.

Логично произвести объединение еще раз, до 2 задач параллельного алгоритма. Однако, выигрыша времени по сравнению с четырехзадачным алгоритмом не получится, длительность расчетов составит 6,16 с. Уменьшение времени коммуникаций в два раза не позволило компенсировать двукратное увеличение времени арифметических операций.

Для произвольного параллельного алгоритма, на этапе объединения, определяя оптимальное число задач L , необходимо искать экстремум функции

$$T(L)=T_a(L)+T_n(L) \quad (3.3),$$

где T - время вычислений по алгоритму на данной вычислительной системе, T_a и T_n - длительность арифметических операций и передач по сети. В рассмотренном примере, при L - степени двойки, $T_a(L) = \frac{13}{100} \frac{64}{L}$ с., $T_n(L)=8$ с. для $L>4$, 4с. для $L=2$ и 2с. для $L=2$.

Учитывая, что отношение скорости производства арифметических операций к скорости передачи данных по сети для стандартных кластерных вычислительных систем колеблется незначительно, синтез алгоритма на этапе объединения не обязательно согласовывать с параметрами конкретной ЭВМ.

Желая сформулировать правила объединения, обратимся к алгоритмам для многомерных разностных уравнений. Существует n стратегий объединения задач, связанных с n -мерной сеточной областью D_h . На двумерной области можно объединять задачи в полосы вдоль одной из координатных осей (одномерное объединение), как на рис. 2.12а, и в блочные области (двумерное объединение), рис.2.12б.

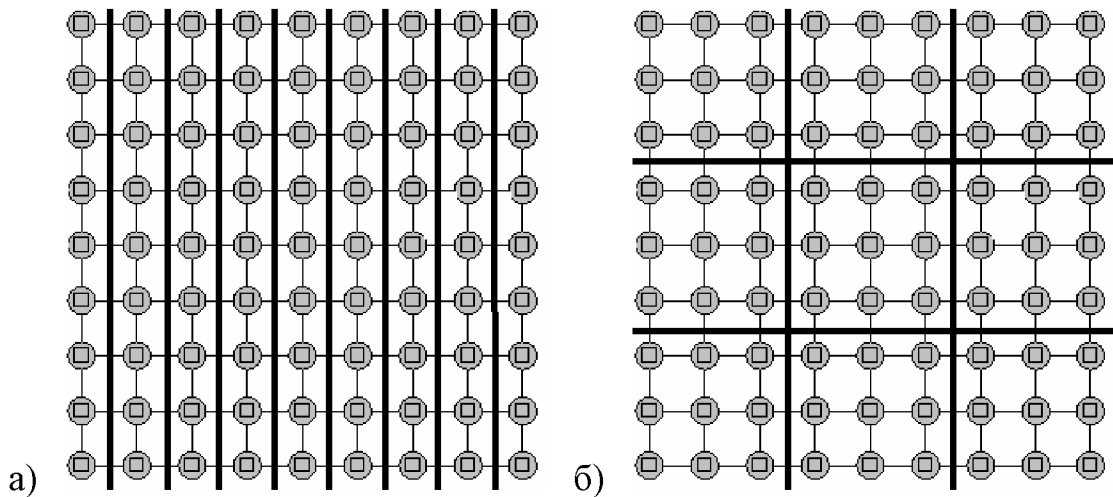


Рис. 2.12 Две стратегии объединения задач, связанных с двумерной сеточной областью: а) - объединение в полосы вдоль координатной оси (одномерное объединение); б) - объединение в блочные области (двумерное объединение).

Проведем сравнение получившихся полосного и блочного алгоритмов. Их задачи порождают вычислительные процессы с одинаковым количеством арифметических операций, но разной структурой коммуникаций. Исследуем коммуникации тех задач алгоритмов, что связаны каналами с максимальным числом соседей (время вычислений по алгоритму будет определяться такими задачами). Для полосного алгоритма это любая задача, кроме первой и последней, для блочного - задача, связанная с центральным блоком сеточной области. Характерная задача полосного алгоритма имеет двух соседей, блочного - четырех.

Для сеточной области $D_h = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=0, 1, \dots, 8, y_j = jh_y, j=0, 1, \dots, 8\}$ предпочтение следует отдать полосному алгоритму, вычисления по нему будут характеризоваться передачей и приемом 4 пакетов по сети, блочный алгоритм использует коммуникации с передачей 8 пакетов.

Рассмотрим квадратную сетку $D_h = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=0, 1, \dots, I, y_j = jh_y, j=0, 1, \dots, I\}$ такую, что для заданного I время передачи сообщения по сети будет характеризоваться в большей мере размером посылки, чем объемом пакета. Для достаточно больших I , когда объем передаваемой информации многократно превышает емкость пакета,

зависимость $T(V)$ на рис. 2.10 можно считать линейной (одно сообщение передается с помощью большого количества пакетов). Предположим, что на этапе объединения, I^2 задач уже сведено в 81 задачу (рис. 2.12). Располагая вычислительной системой с 9-ю процессорами произведем объединение еще раз, остановившись на одном из вариантов: одномерном или двумерном. При одномерном объединении, характерная задача будет посылать I значений сеточной функции правому соседу, I значений соседу слева и столько же принимать от них, оперируя на своих коммуникациях с $4I$ числами. Задача блочного алгоритма, связанная с центральным блоком, посылает каждому из четырех соседей $I/3$ значений функции U_h , организуя посылку и прием $8I/3$ чисел. Двумерное объединение, в данном примере, характеризуется пересылкой существенно меньшего объема данных ($2/3$ от полосного алгоритма) и является более приемлемым.

Когда речь идет о синтезе параллельного алгоритма, чаще всего подразумевается работа с большим объемом вычислений и данных (иначе удовлетворяются известным алгоритмом, реализованным на персональной ЭВМ). Следовательно, производя объединение на области размерности n , отдают предпочтение n - мерному объединению над $n-1$ - мерным, $n-1$ - мерному над $n-2$ - мерным так далее. Эта рекомендация, по форме областей объединения трехмерном случае, получила название правила перехода от поверхности к объему. На рис.2.13 представлен переход от объединения в область, протяженную вдоль одной координатной оси - прямую (одномерное объединение, наименее предпочтительное), через область, протяженную вдоль двух осей - поверхность (двумерное объединение) к области, протяженной вдоль трех осей - объему (трехмерное объединение, наиболее предпочтительное).

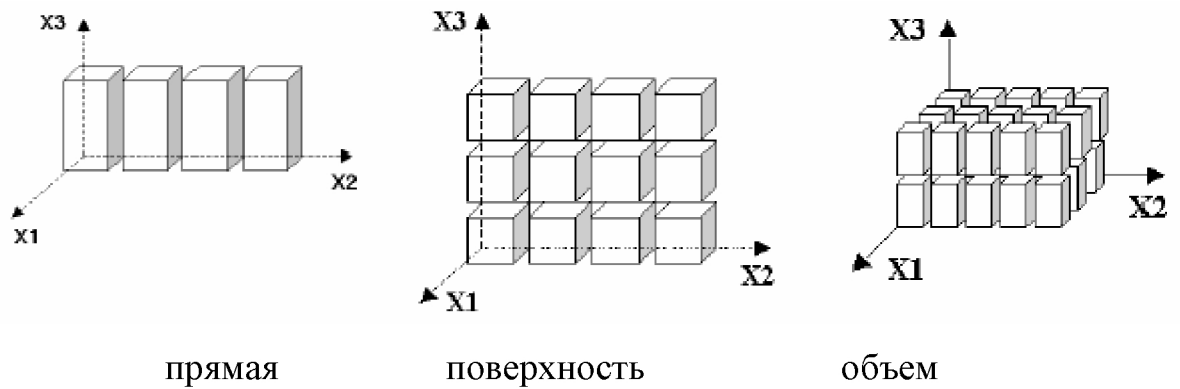


Рис. 2.13 Трехмерная иллюстрация правила перехода от поверхности к объему.

Для прямоугольной сетки с разной дискретизацией правило перехода от поверхности к объему применяют с осторожностью. Например, если $D_h = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=0, 1, \dots, I, y_j = jh_y, j=0, 1, \dots, J\}$, где $I \gg J$, его использование неуместно. Задача полосного алгоритма на рис. 3.14а характеризуется пересылкой $12J$ значений сеточной функции, в то время, как задача блочного алгоритма (рис. 2.14б) пересылкой $4J/3 + 4I/3$ значений. При $I=9J$ блочный алгоритм проигрывает полосному в 3 раза по объему пересылок.

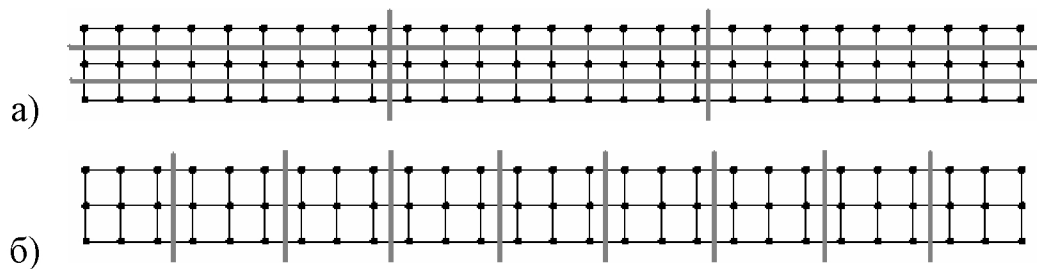


Рис. 2.14 Варианты объединения для прямоугольной сетки с разной дискретизацией. а) - двумерное объединение, б) - одномерное.

Найдем такие параметры дискретизации I, J двумерной сеточной области и количество задач параллельного алгоритма L , при которых блочный алгоритм уступает полосному. Объединение задач в полосном алгоритме разумно проводить вдоль оси с меньшей дискретизацией (так минимизируется объем пересылок), пусть это будет ось Y с параметром дискретизации J . Тогда полосной алгоритм будет характеризоваться

пересылкой $4J$ чисел. В блочном алгоритме организуется пересылка $\frac{4}{\sqrt{L}}J$ чисел по направлению Y и $\frac{4}{\sqrt{L}}I$ чисел по направлению X . При $4J < \frac{4}{\sqrt{L}}J + \frac{4}{\sqrt{L}}I$, то есть для $J < \frac{I}{\sqrt{L}-1}$ полосной алгоритм предпочтительнее блочного. В примере на рис. 2.14 $L=9$ и одномерное объединение становится эффективнее двумерного при $J < I/2$.

На этапе объединения появляются одни задачи, исчезают другие, приходится заново устанавливать коммуникации параллельного алгоритма. При этом топология коммуникаций может претерпеть существенные изменения. Иногда, для сокращения времени вычислений по алгоритму, специально идут на увеличение объема вычислений и устанавливают новые каналы между задачами. Разберем этот прием на примере алгоритма взятия определенного интеграла.

Рассмотрим трехзадачный алгоритм численного интегрирования, каждая задача которого после последнего шага алгоритма должна содержать значение интеграла от функции на всей сеточной области. Коммуникации такого алгоритма состоят из 4 каналов. Пронумеруем каждый канал в соответствии с последовательностью передачи данных по нему (рис. 2.15).



Рис. 2.15 Коммуникации трехзадачного алгоритма численного интегрирования.

Будем рассматривать только те шаги алгоритма, которые следуют за расчетом интегральной суммы (предшествующие шаги не связаны с пересылкой данных и нам не интересны). Задача 3 посылает найденное значение интегральной суммы I_3 задаче 2, которая, прибавляя его к своему значению I_2 , отправляет сумму I_2+I_3 задаче 1. Та, найдя значение интеграла

$I_s=(I_1+I_2+I_3)$ на все сеточной области посылает его задаче 2. В конце алгоритма 2 передает I_s 3. Рассматриваемые шаги алгоритма характеризуются 8 пересылками (4 посылки и 4 приема) и 2 арифметическими действиями, причем как передачи, так и действия происходят последовательно. Время вычислений составит $T=2T_a+8T_{\text{п}}$.

Изменим структуру коммуникаций, связав задачи алгоритма тремя каналами, и увеличим количество арифметических операций в каждой задаче до двух (рис. 2.16).

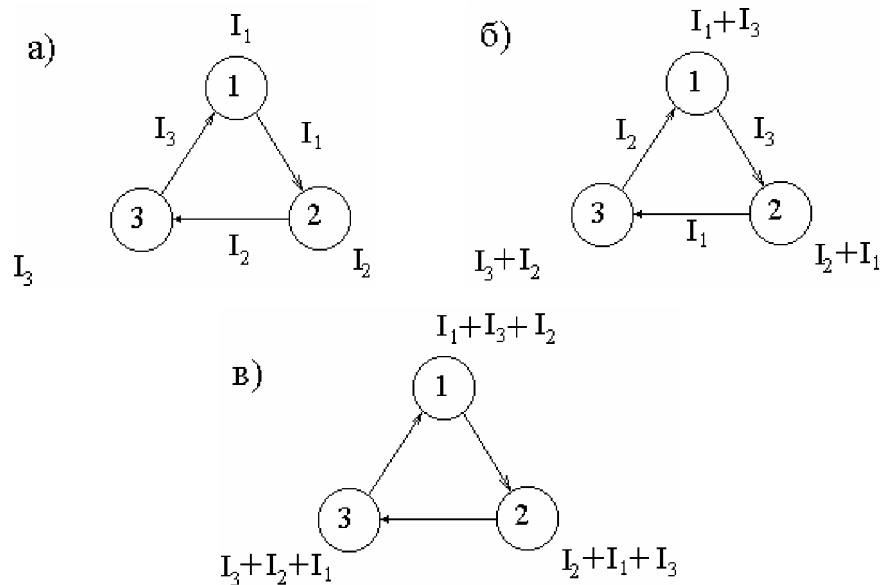


Рис. 2.16 Этапы модифицированного трехзадачного алгоритма численного интегрирования.

Как и в предыдущем случае, ограничимся рассмотрением тех шагов алгоритма, что следуют за вычислением интегральных сумм. На шаге а) задача l , $l=1,...,3$, посылает по выходящему из нее каналу значение I_l и принимает сообщение по входящему каналу (рис. 2.16а). Следующий шаг б) характеризуется суммированием в каждой задаче полученного по каналу и известного ранее значения. Посылается далее значение, полученное на шаге а), принимается новое (рис. 2.16б). На последнем шаге алгоритма каждая задача находит искомое значение I_s определенного интеграла. За три шага произведено 12 передач (на 4 больше, чем в предыдущем случае) и 6 сложений (на 4 операции больше). Но при этом часть сложений и пересылок

осуществлялась параллельно, таким образом, что время арифметических операций осталось тем же, а длительность пересылок данных сократилась вдвое. Учитывая, что 2 операции сложения производятся на несколько порядков быстрее, чем пересылка по сети вычислительной системы 4 пакетов, длительность вычислений, связанных с завершающими этапами алгоритма взятия определенного интеграла, удалось сократить вдвое.

Обратимся к более сложной топологии коммуникаций - дереву. Изучим возможность модификации аналогичного восьмизадачного алгоритма. Время вычислений по заключительным шагам известного алгоритма (рис. 2.9) составит $T=3T_a+12T_n$, производится 7 сложений и 28 пересылок (часть из них параллельно). Первая задача осуществит наибольшее количество сложений (три) и последовательно примет три переданных ей числа (3 приема, 3 посылки). На этапе рассылки значения I_s , до 8-ой задачи ожидаемое число дойдет через 3 посылки и 3 приема (задача 1 посылает I_s задаче 5, та пересылает задаче 7, от нее получает 8-я задача).

Сменим топологию дерева на гиперкуб (2.17).

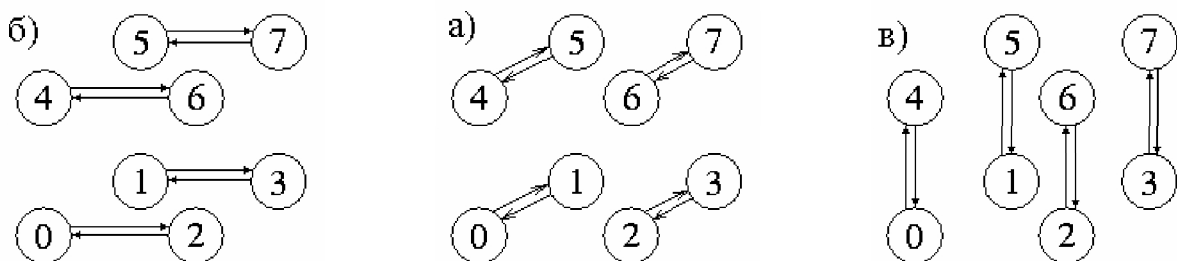


Рис. 2.17 Реализация на гиперкубе восьмизадачного алгоритма взятия определенного интеграл.

На первом шаге модифицированного алгоритма каждая задача получает и передает сообщение от соседа по направлению Z (в правой декартовой системе координат, рис.2.17а). Далее используются каналы по направлениям X (рис.2.17б) и Y (рис.2.17в). С каждым приемом связана одна операция сложения. Следовательно, алгоритм характеризуется 12 сложениями и 48 пересылками. По сравнению с исходным алгоритмом, данный содержит почти вдвое больше арифметических операций и пересылок. Но в отличие от сравниваемого алгоритма, в данном, большая доля сложений и пересылок

исполняется параллельно. Время вычислений по нему составит $T=3T_a+6T_n$, что вдвое меньше времени вычислений по алгоритму на дереве.

Учитывая изложенное, производя синтез параллельного алгоритма на этапе объединения, будем руководствоваться следующими рекомендациями.

1. Укрупняемые задач должны оставаться сбалансированными.
2. Руководствоваться правилом перехода от поверхности к объему. Разумеется только там, где его применение уместно. Понимать это правило как тенденцию, а не закон.
3. При изменении коммуникаций не ограничиваться старой топологией, изучать возможность установления каналов, использование которых позволит сократить время вычислений.
4. Применять прием тиражирования вычислений для уменьшения объема пересылаемых данных.

Коротко остановимся на этапе распределения задач параллельного алгоритма по процессорам вычислительной системы. Данный этап более связан с реализацией алгоритма на конкретной ЭМВ, чем с его синтезом, следовательно, относится к изучаемому студентами курсу "Параллельное программирование" и находится за рамками настоящего пособия. Все же, считаем необходимым осветить характерные особенности этого этапа.

Ошибочно полагать, что при реализации любого алгоритма принято ставить в соответствие одному процессору вычислительной системы только одну задачу параллельного алгоритма. Рассмотрим случай, когда одному процессору можно поручать вычисления по нескольким задачам алгоритма.

Предположим существование алгоритма, задачам которого присущи простои. В некоторый момент времени часть задач такого алгоритма производит вычисления, другая часть ожидает передачи сообщений по каналам и вынужденно простаивает (без получения необходимых данных вычислений не производится). Например, в алгоритме, коммуникации которого изображены на рис. 3.15 задача 3 участвует в вычислениях и

пересылках только на первом и последнем шаге, простаивая во время исполнения остальных шагов алгоритма.

Если на один процессор вычислительной системы приходится две задачи, вычисления по которым одновременно не производятся (одна задача всегда простаивает), то общее время вычислений по алгоритму не возрастет.

При разработке параллельного алгоритма также следует иметь в виду, что современные многопроцессорные вычислительные системы позволяют одновременно на одном процессоре производить вычисления и осуществлять прием или передачу сообщений по сети. Эта особенность будет учтена нами в дальнейшем изложении (например при построении блочного алгоритма перемножения матриц).

В целом, касаясь восходящего метода синтеза параллельных алгоритмов, следует заметить, что последовательность прохождения этапов метода (от разбиения через установку коммуникаций и объединение к распределению по процессорам вычислительной системы) в практике построения алгоритмов выдерживается не всегда. Приходится возвращаться на предыдущие этапы, корректируя количество задач, их содержание и топологию коммуникаций. Синтез алгоритмов процесс творческий, не следует ограничиваться строгим соблюдением последовательности этапов изложенного метода.

2.3. Нисходящий подход к синтезу параллельных алгоритмов.

Как уже отмечалось, нисходящий подход характеризуется разбиением последовательного алгоритма на как можно более крупные слабовзаимодействующие блоки (в литературе такой прием построения параллельных алгоритмов иногда называется блочно-иерархическим методом). Под блоками алгоритма понимают некоторую совокупность инструкций (как при функциональной декомпозиции) или данных (декомпозиция данных).

Иерархия, которую задают размеры блоков, при функциональной декомпозиции строится от арифметических операций (низший уровень иерархии), через все более сложные арифметические выражения до крупных блоков - сегментов алгоритма (высший уровень иерархии).

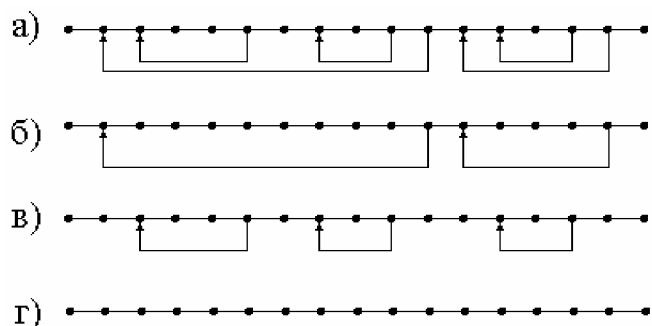


Рис. 2.18 а) схема последовательного алгоритма с разбиением вычислений на блоки; б) высший уровень иерархии, сегменты алгоритма - внешние циклы; в) второй уровень иерархии - вложенные циклы; г) третий уровень иерархии - арифметические выражения

На рис. 2.18 представлена схема последовательного алгоритма с разбиением вычислений на блоки и задаваемой этим разбиением иерархией. Стрелками показаны циклические конструкции, точками - арифметические выражения. Вычисления по схеме производятся слева на право.

При декомпозиции данных иерархическая классификация различает значения отдельных величин (низший уровень), векторов и матриц, многомерных массивов (высший уровень).

Блочно-иерархический метод можно рассматривать как модификацию этапов разбиения и установки коммуникаций восходящего подхода. Отличие заключается в выборе критерия окончания разбиения. Если в восходящем подходе разбиение заканчивается на самом мелком характерном фрагменте последовательного алгоритма, то применение рассматриваемого метода подразумевает проверку времени исполнения вычислений по алгоритму каждый раз после разбиения на более мелкие блоки при движении вниз по иерархической структуре вычислений или данных. При движении к иерархическим структурам низкого уровня увеличивается число задач

параллельного алгоритма, большее количество арифметических операций выполняется одновременно, что сокращает длительность вычислений, связанных с исполнением арифметических операций. Однако, при этом усложняется топология коммуникаций, задачам приходится обмениваться большим объемом данных - это, начиная с определенного уровня иерархии, увеличивает общую длительность вычислений. Процесс разбиения в нисходящем подходе оканчивается на том иерархическом уровне m_{opt} , для которого время вычислений T по алгоритму характеризуется наименьшим значением t_{min} (рис. 2.19).

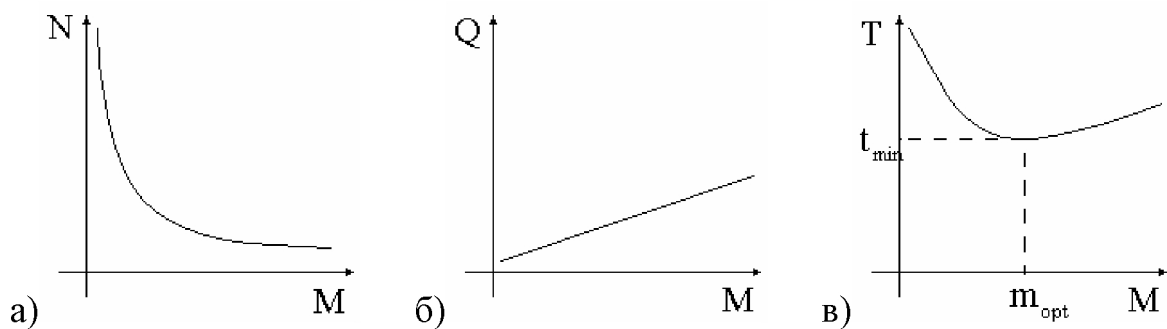


Рис. 2.19 Определение оптимального уровня блочной иерархической структуры при разбиении. а) - зависимость количества арифметических операций N , производимых одной задачей параллельного алгоритма от уровня иерархии M ; б) - зависимость от уровня иерархии объема данных Q , принимаемых и посылаемых по каналам одной (характерной) задачей алгоритма; в) - аналогичная зависимость для времени вычислений по параллельному алгоритму.

Проиллюстрируем применение блочно-иерархического метода на примере построения параллельного алгоритма перемножения квадратных матриц /4/. Даны две матрицы A и B размерностью $n \times n$. Элементы результирующей матрицы $C=A \times B$ находятся по формуле $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$. На рис. 2.20 схематично записан последовательный алгоритм матричного перемножения.

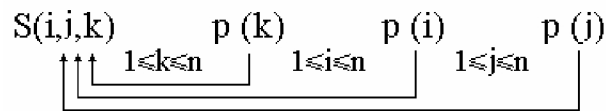


Рис. 2.20 Блочнo-иерархическая структура вычислений по последовательному алгоритму матричного перемножения. $S(i,j,k)$ - операция суммирования произведений $a_{ik}b_{kj}$; цикл $p_1(k)$ используется для вычисления одного элемента матрицы C ; цикл $p_2(i)$ перебирает элементы матрицы C по строкам, цикл $p_3(j)$ по столбцам.

Осуществляя функциональную декомпозицию разобьем на блоки цикл p_3 - верхний уровень иерархии вычислений (рис. 2.21). Количество задач L параллельного алгоритма для простоты выберем таким, чтобы n делилось на L без остатка.

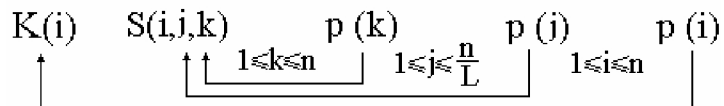


Рис. 2.21 Структура вычислений по параллельному алгоритму перемножения матриц без обмена данных между задачами.

Данные между задачами алгоритма распределим так, как это показано на рис. 2.22.

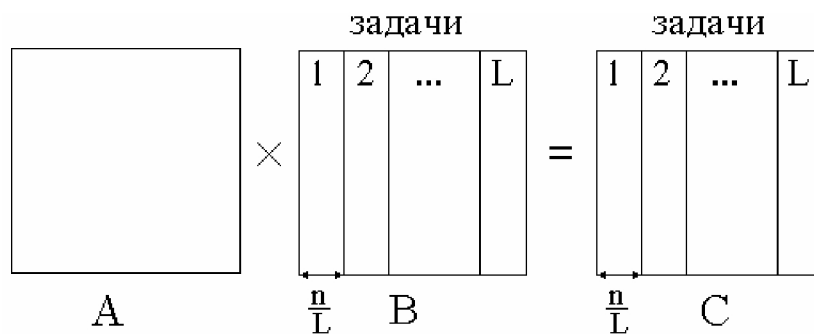


Рис. 2.22 Распределение данных между задачами алгоритма матричного перемножения без коммуникаций.

Каждая задача алгоритма целиком хранит матрицу A и вертикальные полосы, состоящие из столбцов матриц B и C , шириной n/L . Все данные, необходимые для вычисления любого элемента из полосы матрицы C , принадлежащей определенной задаче, наличествуют в памяти этой задачи.

Таким образом, вычисления по алгоритму не связаны с установкой коммуникаций и обменом данными.

Часто при построении и исследовании параллельных алгоритмов рассматривают начальный этап, связанный с распределением данных. На этом этапе первая задача алгоритма считывает из файла начальные данные и распределяет их между остальными задачами. Здесь и далее будем предполагать, что рассматриваемые алгоритмы входят в качестве модулей в более сложные алгоритмы, распределение данных в которых организовано таким образом, что на всех этапах (кроме первого и последнего) дополнительных пересылок, связанных с начальным распределением данных, не требуется.

Предложенный алгоритм матричного перемножения редко используется на практике из-за больших затрат памяти вычислительной системы. За увеличение скорости вычислений в L раз приходится платить увеличением объема занимаемой памяти (матрица A копируется L раз).

Запишем менее затратный алгоритм, изменив разбиение области данных (рис. 2.23).

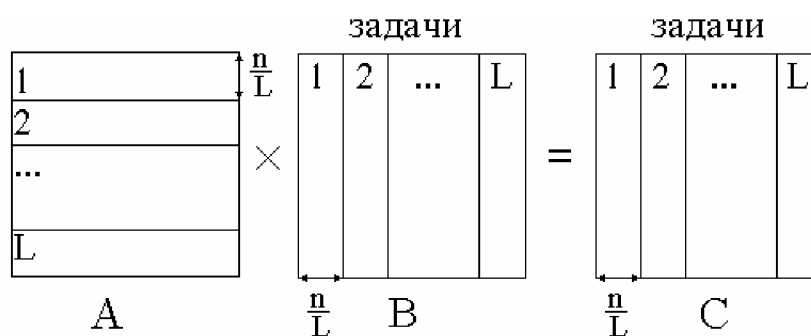


Рис. 2.23 Распределение матриц между задачами параллельного алгоритма матричного перемножения

Матрица A разделена между задачами на вертикальные полосы шириной n/L . При новой декомпозиции данных изменится структура вычислений по новому алгоритму (рис. 2.24).

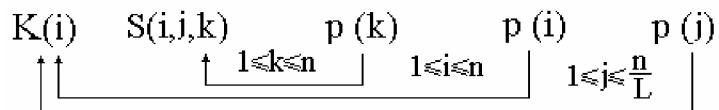


Рис. 2.24 Структура вычислений по параллельному алгоритму

матричного перемножения с одномерной декомпозицией области данных.

Операция $K(i)$ - пересылка i - ой строки матрицы A от задачи, в памяти которой эта строка содержится, всем остальным задачам алгоритма.

Первая задача перед вычислением элемента c_{11} разошлет первую строку матриц A остальным задачам. Второй задаче эта строка понадобится для нахождения элемента $c_{1, n/L+1}$, третьей для нахождения $c_{1, 2n/L+1}$, ..., последней для $c_{1, \frac{(L-1)n}{L}+1}$. Задачи со второй по последнюю не содержат в

своей памяти первую строку матрицы A и вынуждены принимать ее от первой задачи. Строку A под номером $n/L+1$ будет рассылать вторая задача, а задачи 1, 3, 4, ..., L эту строку принимают. И так далее, до последней строки матрицы, рассылкой которой займется последняя задача.

В новом алгоритме на вычисление одного элемента матрицы C приходится посылка одной строки матрицы A . На n умножений и $n-1$ сложение - пересылка n чисел. Независимо от характеристик вычислительной системы, на которой будут производиться вычисления по данному алгоритму, можно с уверенностью сказать, что большую часть времени ветви параллельного процесса будут обмениваться данными, а меньшую - производить арифметические операции. Время расчетов по представленному алгоритму может оказаться больше, чем по последовательному на однопроцессорной ЭВМ, если n/L достаточно велико.

В иерархической конструкции на рис. 2.24 переставим местами циклы $p_2(i)$ и $p_3(j)$ (рис. 2.25).

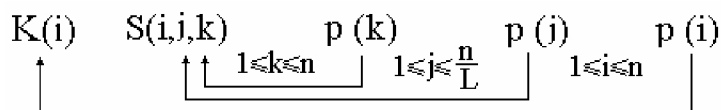


Рис. 2.25 Структура вычислений по модифицированному параллельному алгоритму с одномерной декомпозицией области данных.

Такая перестановка допустима и в последовательном алгоритме, следовательно, иерархический принцип разбиения на блоки соблюдается.

Вычисляя элементы одной строки матрицы C (а значит все элементы одной строки в вертикальной полосе матрицы C , принадлежащей каждой задаче) необходимо передать соответствующую строку матрицы A только один раз. После передачи n чисел вычисляется n/L элементов C с использованием n^2/L умножений и $n(n-1)/L$ сложений. При таком соотношении объема передач и арифметических операций, ветви параллельного вычислительного процесса, на некоторых вычислительных системах, большую часть времени вычислений будут уделять производству арифметических операций, а меньшую - передаче по сети строк матрицы A . Модифицированный алгоритм обеспечит сокращение длительности вычислений по сравнению с последовательным случаем.

2.4. Характеристики параллельных вычислительных процессов и алгоритмов.

В предыдущем параграфе за меру эффективности параллельного алгоритма принималось время работы порожденного этим алгоритмом параллельного вычислительного процесса (ПВП). Действительно, при прочих равных показателях (количеству задач, максимальному числу соседей одной задачи, объему занимаемой памяти, точности полученного решения) предпочтительнее тот алгоритм, вычисления по которому производятся быстрее. Однако, ПВП всегда связан с вычислительной системой, на которой протекает. Время его работы определяется быстродействием процессоров, конфигурацией сети, скоростью доступа к оперативной памяти и т.п. Сравнения двух алгоритмов на разных вычислительных системах могут дать

разные результаты. Следовательно, говоря об эффективности необходимо различать эффективность ПВП и алгоритмов.

Время вычислений, являясь абсолютной величиной, не всегда удобно для сравнения двух алгоритмов или процессов, когда принято пользоваться формулировками "во сколько раз" и "ограничено величиной". В теории параллельных вычислений, сравнивая алгоритмы или процессы, пользуются относительными характеристиками: ускорением, эффективностью и масштабируемостью. Им и посвящен настоящий параграф.

2.4.1. Характеристики параллельных вычислительных процессов.

Измерив длительность вычислений на однопроцессорной ЭВМ и оставшись недовольным, исследователь может задаться вопросом: во сколько раз ускорятся вычисления, окажись в его распоряжении вычислительная система, состоящая из N процессоров? Говоря языком теории параллельных вычислений, его интересует ускорение ПВП S , определяемое как отношение времени T^1 исполнения последовательного процесса ко времени T^N исполнения ПВП на N процессорах /5/: $S = \frac{T^1}{T^N}$ (2.4). Эффективностью ПВП

называется производная от ускорения величина /5/ $E = \frac{S}{N} = \frac{T^1}{NT^N}$ (2.5).

Неопытный исследователь ожидает увидеть ускорение равным числу процессоров. Чаще всего, на практике, S значительно ниже N .

Остановимся на двух причинах этого явления, связав с ними два подхода к оценке ожидаемого ускорения ПВП. Первая причина кроется в структуре последовательного вычислительного процесса. Любой такой процесс содержит часть, которую нельзя исполнить параллельно, разделив между несколькими процессорами вычислительной системы. Существование такой части может быть связано с человеческим фактором (оператор последовательно вводит начальные данные, последовательно просматривает результаты), с особенностями алгоритма, порождающего данный процесс, с

наличием "неделимых" операций (сложения, умножения, чтения одного числа из файла и т.п.). Обозначим время исполнения всех нераспараллеливаемых действий последовательного процесса за t_n (рис. 2.26).

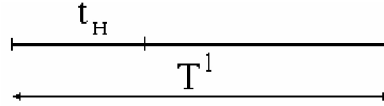


Рис. 2.26 Структура последовательного вычислительного процесса.

Тогда $f = \frac{t_n}{T^1}$ есть доля длительности таких действий во времени всего

процесса. При этом $T^N = fT^1 + (1-f)\frac{T^1}{N}$, где первое слагаемое - длительность последовательной части ПВП, второе - время исполнения параллельной части. Ускорение $S = \frac{1}{f + \frac{1-f}{N}}$ (2.6).

Формула (2.6) выведена в предположении, что ветви ПВП не взаимодействуют друг с другом, что в общем случае неверно. Обмен данными увеличивает длительность ПВП, поэтому (2.6) следует обратить в неравенство $S \leq \frac{1}{f + \frac{1-f}{N}}$ (2.7), известное, как закон Амдала (по фамилии автора, опубликовавшего его в 1967 году).

Закон Амдала ограничивает сверху ускорение ПВП величиной, зависящей от количества процессоров вычислительной системы и структуры последовательного вычислительного процесса. При $N \rightarrow \infty$ $S \leq 1/f$, для идеально распараллеливаемого последовательного процесса $S \leq N$ (рис. 2.27).

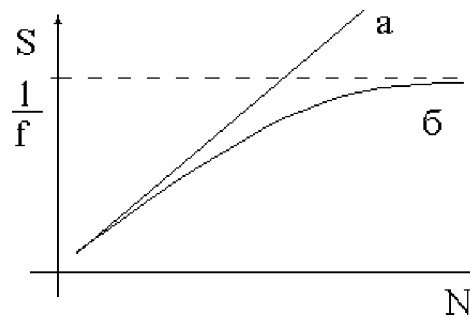


Рис. 2.27 ограничения, налагаемые на ускорение ПВП законом Амдала.

Кривая а - ускорение при $f=0$; б - при $f>0$.

Если ускорение $S=N$ принять за максимальное, то эффективность E показывает долю достигнутого ускорения от максимального.

Проиллюстрируем изложенное численными примерами. Пусть в нашем распоряжении имеется вычислительная система, состоящая из 1000 процессоров, а структура последовательного вычислительного процесса характеризуется величиной $f=0,001$ (нераспараллеливаемый фрагмент длится всего 0,1% от времени процесса). Тогда $S \approx 500$ и $E \approx 0,5$. Использование 1000 процессоров привело к ускорению всего в 500 раз. Для $f=0,01$ ускорение и эффективность снизятся до значений $S \approx 91$ и $E \approx 0,091$. Даже малые значения f при большом количестве процессоров серьезно ограничивают ускорение ПВП, снижая его эффективность.

Второй подход к оценке эффективности ПВП связан с учетом времени обмена сообщениями $t_{об}$ между ветвями процесса и времени простоев $t_{пр}$ ветвей, ожидающих приема сообщений. Синтезируя параллельные алгоритмы $t_{пр}$ стремятся свести к нулю, что не всегда получается (например при решении СЛАУ методом исключений Гаусса). Дальнейшие рассуждения верны в предположении, что ветви ПВП совершенно сбалансированы: все выполняют одинаковое количество арифметических операций (длительностью $t_{ар}$), обмениваются одинаковым объемом данных и вынуждены простаивать одно и то же время. При этом в один момент времени ветви разные ветви могут производить отличные действия. В рамках данного подхода предполагается, что кроме арифметических операций, обменов и простоев других действий не производится. Тогда $T^1 = t_{ар}N$,

$$T^N = t_{ap} + t_{об} + t_{пр}, \text{ а } E = \frac{t_{ap}}{t_{ap} + t_{об} + t_{пр}} \quad (2.8).$$

Эффективность показывает долю, приходящуюся на время арифметических операций в длительности ПВП. Чем меньше времени ПВП тратит на обмены и простои, и больше на арифметические операции - тем он эффективнее. Зная длительности последовательного и параллельного процессов можно выразить совокупное время обменов и простоев $t_{об} + t_{пр} = T^N - T^1/N$.

В данном подходе раскрывается иной смысл ускорения. Величина $N \times E = S$ есть количество процессоров (из N), занятых исключительно арифметическими операциями; $N(1-E) = N - S$ - количество процессоров, занятых только обменами и простоями. Эти числа условны, мы сводим все арифметические вычисления в одну группу процессоров (к тому же с нецелым числом таких процессоров) в то время, как они распределены по всем процессорам вычислительной системы.

Если график зависимости ускорения от числа процессоров стагнирует, начиная с некоторого N' , значит добавленные $N - N'$ процессоров не производят полезных действий и их использование излишне (рис. 2.28).

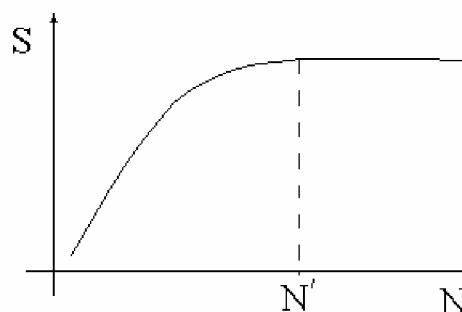


Рис 2.28 Определение оптимального числа процессоров для реализации ПВП.

$$\text{Ускорение оценивается величиной } S \leq \frac{N t_{ap}}{t_{ap} + t_{об} + t_{пр}} = \frac{N}{1 + k} \quad (2.9), \text{ где}$$

$$k = \frac{t_{об} + t_{пр}}{t_{ap}}.$$

На практике не удастся добиться идеальной балансировки ветвей ПВП, этим объясняется знак неравенства в оценке (2.9). При $t_{об} + t_{пр} \rightarrow 0$ ускорение стремится к максимальному значению.

Во втором подходе к оценке ускорения ПВП (так же, как и в первом) учитывается структура последовательного процесса. Величина f не входит в явном виде в выражение (2.9), но k является функцией от f . Параллельные ветви или дублируют нераспараллеливаемые вычисления (тогда $t_{ap}=t_{ap}(f)$), или ожидают, пока одна из ветвей производит такие вычисления ($t_{np}=t_{np}(f)$).

На практике допустимо использование обеих оценок ускорения: (2.7), если ПВП состоит из слабовзаимодействующих ветвей и (2.9), когда величину k можно оценить до реализации ПВП.

Зависимость ускорения от количества процессоров вычислительной системы не является исчерпывающей характеристикой семейства ПВП, порожденных одним алгоритмом. При увеличении объема данных, которым оперирует ПВП, значения f и k могут уменьшаться, ускорение на том же числе процессоров - увеличиваться. Изучая это свойство ПВП, исследователь иногда строит много графиков зависимостей $S(N)$ для разного объема данных (рис. 2.29).

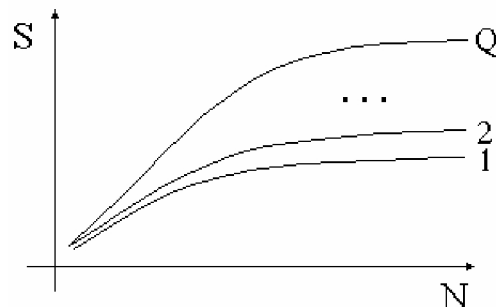


Рис. 2.29 Зависимости $S(N)$ для разного объема данных V ПВП. $V=V_1$ (кривая 1), $V=V_2$ (кривая 2), ..., $V=V_Q$ (кривая Q). $V_1 < V_2 < \dots < V_Q$.

В теории параллельных вычислений принято пользоваться другой величиной - масштабируемостью M . $M = \frac{V^N}{V^1}$ при $T^1(V^1) = T^N(V^N)$.

Масштабируемость ПВП есть отношение объема данных V^N параллельного процесса к объему данных V^1 последовательного процесса. Длительность обоих процессов, используемых для определения масштабируемости, должна совпадать. Масштабируемость показывает во сколько раз увеличиться объем

данных при реализации ПВП на N процессорах по сравнению с однопроцессорной ЭВМ, если время реализации останется постоянным.

2.4.2 Характеристики параллельных алгоритмов.

Говоря об алгоритмах, в отличие от вычислительных процессов, мы не можем оперировать временем работы, выраженном в секундах, физическим временем обмена сообщениями, простоев и арифметических операций. В нашем распоряжении будет количество арифметических операций, исполняемых характерной задачей алгоритма, объем посылаемых и принимаемых ею данных. Простой некоторой задачи обычно связаны с арифметическими действиями другой задачи, результаты вычислений которой ожидает некоторая, и выражаются через количество соответствующих операций другой задачи.

Следовательно, ускорение сбалансированного алгоритма

$$S = \frac{L \times \tau_{ap}}{\tau_{ap} + \tau_{об} + \tau_{пр}} \quad (2.10), \text{ где } \tau_{ap} - \text{"условная" длительность арифметических}$$

операций характерной задачи алгоритма, выраженная через количество операций сложения, умножения, деления и время одной такой операции (τ_x, τ_+, τ_+); $\tau_{об}$ - "условная" длительность обменов алгоритма, выраженная через объем пересылаемых данных и время пересылки по каналу одного числа (τ_k); L - число задач алгоритма.

По аналогии с эффективностью ПВП, за эффективность параллельного алгоритма примем величину $\eta = \frac{\tau_{ap}}{\tau_{ap} + \tau_{об} + \tau_{пр}} \quad (2.11)$, показывающую долю

времени, затрачиваемого на арифметические операции во всей длительности вычислений. В литературе встречается другое определение эффективности

алгоритма: $\delta = \frac{\tau_{об} + \tau_{пр}}{\tau_{ap}} \quad (2.12)$, обратное предыдущему. Отличие связано с

разными областями применения критериев η и δ .

Рассмотрим параллельный алгоритм матричного перемножения из 2.3. Будем увеличивать число задач алгоритма, не меняя размерности матриц. Объем пересылок при этом останется постоянным. Каждая задача отправляет остальным свою часть матрицы A и принимает от них все другие части этой матрицы. С ростом L уменьшается ширина полосы матрицы A , принадлежащей каждой задаче (рис. 2.32), но объем пересылок, в которых участвует задача не изменяется. Уменьшается количество посылаемых задач чисел, увеличивается количество принимаемых (в сумме они составляют объем матрицы A). Общее время вычислений по алгоритму сокращается, эффективность по критерию δ возрастает, по критерию η падает. Если более эффективным алгоритмом считать тот, вычисления по которому производятся быстрее, то критерий η в данном случае, применять нельзя.

Решая этот парадокс откажемся от общего понятия эффективности параллельного алгоритма, рассматривая отдельно эффективность по времени (чем быстрее производятся вычисления, тем алгоритм эффективнее) и эффективность по вычислениям (чем выше доля арифметических операций в общем количестве действий, тем лучше). Для алгоритма матричного перемножения из 2.3. эффективность по времени вычисляется по критерию δ , по вычислениям - с помощью критерия η .

На рис. 2.30 предложен другой пример, для которого необходимо подобрать критерии эффективности при сравнении двух алгоритмов, представленных временами арифметических операций и обменов с простоями.

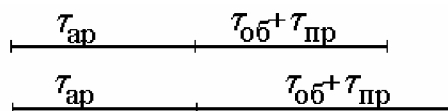


Рис. 2.30 Соотношение длительности арифметических операций, обменов и простоев двух сравниваемых алгоритмов.

Если в качестве критерия эффективности по времени опять использовать δ , то первый алгоритм, содержащий больше обменов и

простоев при равном времени арифметических операций окажется более эффективным, что неверно. Здесь, для оценки эффективности по времени, уместен критерий η .

Подбирая критерий эффективности по времени, сравнивая два разных алгоритма или один алгоритм с варьирующимся числом задач, будем руководствоваться следующими соображениями. При одинаковой для обоих алгоритмов величине $\tau_{ар}$ применяем критерий η (эффективней по времени будет алгоритм, задачи которого который меньше обмениваются данными и простаивают). Если у двух алгоритмов одинаковы времена $\tau_{об} + \tau_{пр}$, верен критерий δ (эффективней по времени алгоритм, производящий меньше арифметических вычислений). Если все "условные" времена для обоих алгоритмов различны, остается сравнивать напрямую длительность вычислений по алгоритму $\tau_{ар} + \tau_{об} + \tau_{пр}$, отказываясь от обоих критериев эффективности.

Определяя эффективность по вычислениям пользуются исключительно критерием η ., как точно соответствующим определению такой эффективности.

Хочется обратить внимание читателей на то, что в настоящем параграфе рассматривалась эффективность параллельной реализации вычислительного процесса и эффективности распараллеливания алгоритма. Под эффективностью не понималась точность полученного ПВП численного решения, его сходимость к аналитическому, корректность и т.д. Эффективно распараллеленный алгоритм может обладать худшими характеристиками по сравнению с неэффективно распараллеленным, когда речь идет о точности полученного решения, порядке его сходимости и т.д.

ГЛАВА 3 ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ РЕШЕНИЯ РАЗНОСТНЫХ УРАВНЕНИЙ.

В настоящей главе излагается методика построения параллельных алгоритмов решения разностных уравнений простейших явной и неявной схем. Для определенности рассматриваются схемы, применяемые при численном решении нестационарного линейного однородного уравнения теплопроводности $\frac{\partial U}{\partial t} - a\Delta U = 0$, где $U = U(\bar{x}, t)$ - температура среды,

характеризующийся температуропроводностью a ; t - время; $\bar{x}$ - пространственные координаты. На границе области на функцию U налагаются условия первого рода.

Описание алгоритмов ограничивается пояснением расчета сеточных функций на одном временном слое. Вычисления на других слоях полностью аналогичны. Все характеристики параллельных алгоритмов расчета на одном слое верны для произвольного количества слоев, так как длительность вычислений τ^q на q слоях линейно зависит от числа слоев и времени расчета сеточных функций на одном слое τ^1 ($\tau^q = q \times \tau^1$).

Представленные алгоритмы могут использоваться для решения различных уравнений математической физики методами конечных разностей и Галеркина.

3.1. Параллельные алгоритмы решения сеточных уравнений явных разностных схем.

3.1.1 Параллельный алгоритм для явных разностных схем, одномерный случай.

Решая одномерное (по пространству) уравнение теплопроводности

$\frac{\partial U}{\partial t} - a \frac{\partial^2 U}{\partial x^2} = 0$ используем простейшую явную схему

$\frac{U_i^{n+1} - U_i^n}{h_t} - a \frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{h_x^2} = 0$, записанную на равномерной сетке

$\omega^1 = \{(t_k, x_i), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=1, \dots, I=L/h_x+1\}$. Такая схема позволяет выразить значение сеточной функции U_i^{n+1} на новом временном слое $n+1$ через соседние функции $U_{i-1}^n, U_i^n, U_{i+1}^n$ на предыдущем слое n в явном виде: $U_i^{n+1} = a \frac{h_t}{h_x^2} (U_{i-1}^n - 2U_i^n + U_{i+1}^n) + U_i^n$ (3.1).

Построение параллельного алгоритма для вычислений по (3.1) на ω^1 начнем с декомпозиции сеточной области (рис. 3.1б).

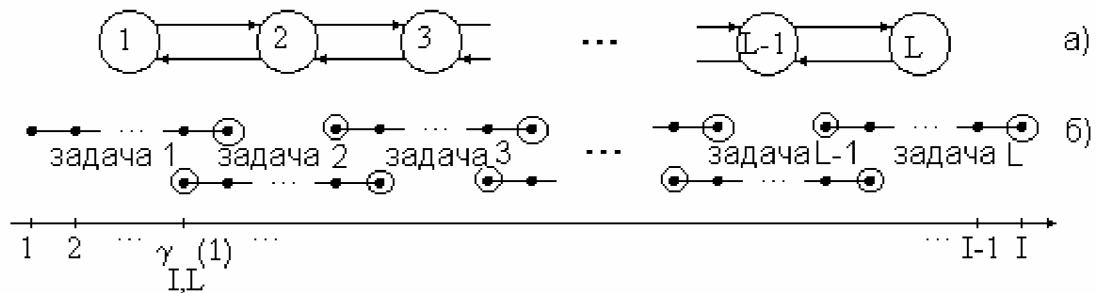


Рис. 3.1 Коммуникации алгоритма для одномерного случая а) и разбиение сеточной области б). Кружком обведены узлы сетки, в которых сеточная функция не вычисляется, но используется.

Задаче l ($2 \leq l \leq L-1$) параллельного алгоритма принадлежит $\gamma_{L,L}(l)+2$ значений сеточных функций и узлов сетки, ограниченных фрагментом области ω^1 с индексами r_1^l-1, r_2^l+1 . Первая задача содержит $\gamma_{L,L}(1)+1$ значений в пределах $1 \leq i \leq r_2^1+1$, последняя задача столько же значений в пределах $r_1^L-1 \leq i \leq I$.

На первом шаге алгоритма задача l ($1 \leq l \leq L$) вычисляет значения U_i^{n+1} для $r_1^l \leq i \leq r_2^l$ по формуле (3.1). При расчете $U_{r_1^l}^{n+1}$ все задачи, кроме первой, используют значение $U_{r_1^l-1}^n$, для хранения которого дублируется предпоследний узел задачи $l-1$. Вычисляя $U_{r_2^l}^{n+1}$ все задачи, кроме последней,

используют значение $U_{r_2^1+1}^n$, для хранения которого дублируется второй узел задачи $l+1$. Поэтому на рис 3.1б перекрываются фрагменты соседних подобластей ω^1 , принадлежащие разным задачам. Первая и последняя задача не нуждаются в значениях U_0^n и U_{l+1}^n , их подобласти меньше на один узел.

Второй шаг алгоритма связан с его коммуникациями (рис.3.1а). Каждой задаче l ($2 \leq l \leq L-1$) на шаге $n+2$ понадобятся значения $U_{r_2^1+1}^{n+1}, U_{r_1^1-1}^{n+1}$, которые следует загодя получить от задач $l+1, l-1$. Первая задача заинтересована в получении $U_{r_2^1+1}^{n+1}$ от задачи 2, последняя в получении $U_{r_1^1-1}^{n+1}$ от задачи $L-1$. Пересылку данных осуществим в два приема. Сначала задачи l ($1 \leq l \leq L-1$) посылают правым соседям значения $U_{r_2^1}^{n+1}$ и принимают (кроме первой задачи) от левых соседей $U_{r_1^1-1}^{n+1}$. Задача L принимает от $L-1$ $U_{r_1^1-1}^{n+1}$. Затем задачи l ($2 \leq l \leq L$) посылают левым соседям значения $U_{r_1^1}^{n+1}$ и принимают (кроме последней задачи) от правых соседей $U_{r_2^1+1}^{n+1}$. Первая задача принимает от второй $U_{r_2^1+1}^{n+1}$.

Такой порядок пересылок позволяет реализовывать вычисления по алгоритму не только на кластерах, но и на векторных и конвейерных ЭВМ, для которых пересылки данных производятся всеми задачами одновременно, в одном направлении и только соседям (левому или правому). Однотипность вычислений в каждой задаче и наличие всего двух соседей (для первой и последней задач сосед один) позволяют отнести алгоритм к классу векторных и конвейерных алгоритмов. Под собственно параллельными алгоритмами в литературе обычно понимают исключительно те алгоритмы, вычисления по которым реализуются на ЭВМ с кластерной архитектурой.

Оценивая ускорение алгоритма, длительность арифметических операций по нему положим равной $\gamma_{I,L}(l)\tau_1$, где $\tau_1=3\tau_+ + \tau_\times$ время расчета одной сеточной функции по формуле (3.1).

Определяя значение функции γ коснемся проблемы распределения данных между задачами параллельного алгоритма. Предположим существование одномерного массива размерности I . Необходимо распределить все его элементы между L задачами ($L \leq I$) как можно более равномерно, сохраняя балансировку задач. Но так, чтобы никакие две задачи не содержали элементы распределяемого массива с одинаковыми порядковыми номерами (исключить дублирование). Если I не делится на L

без остатка, используют функцию $\gamma_{I,L}(l) = \begin{cases} \left\lceil \frac{I}{L} \right\rceil + 1, & \text{при } I \% L \geq l \\ \left\lfloor \frac{I}{L} \right\rfloor, & \text{при } I \% L < l \end{cases}$, где l -

номер задачи алгоритма ($1 \leq l \leq L$); $\lceil \cdot \rceil$ - операция округления до ближайшего меньшего целого; с помощью бинарной операции $a \% b$ находится остаток от деления a на b , $I \% L = I - \left\lfloor \frac{I}{L} \right\rfloor \times L$.

Задаче с номером l отойдет $\gamma_{I,L}(l)$ элементов распределяемого массива, от элемента под номером $\sum_{g=1}^{l-1} \gamma_{I,L}(g) + 1$ до элемента $\sum_{g=1}^l \gamma_{I,L}(g)$.

Коэффициент $\alpha \frac{h_t}{h_x^2}$ вычисляется в начале алгоритма и далее

используется как константа, поэтому операции деления и умножения, с ним связанные, в τ_1 не входят. При смене дифференциального шаблона будем подставлять в выражение $\gamma_{I,L}(l)\tau_1$ другое значение τ_1 . Например, для шаблонов на рис. 3.2 τ_1 увеличится, $\gamma_{I,L}(l)$ останется прежним.

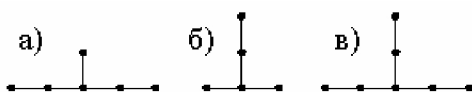


Рис. 3.2 Различные дифференциальные шаблоны для построения разностных схем.

В рассматриваемом алгоритме все пересылки связаны с передачей или приемом одного числа (значения сеточной функции), следовательно, длительность одного приема или передачи равна τ_n . Время всех пересылок алгоритма равно совокупному времени передач и приемов одной задачи 1 ($2 \leq L \leq L-1$) и составит $4\tau_n$ (два приема и две передачи). Применяя алгоритм для решения разностных уравнений, составленных по шаблонам на рис.3.2а,в получим длительность пересылок, равную $8\tau_n$, по шаблону на рис.3.2б - $4\tau_n$.

$$S = \frac{I \times \tau_1}{\gamma_{I,L}(I) \times \tau_1 + 4\tau_n} = \frac{I}{\gamma_{I,L}(I) + 4k_1} \xrightarrow{L \rightarrow I} \frac{I}{1 + 4k_1} \approx \frac{I}{4k_1}, \quad \text{где } k_1 = \frac{\tau_n}{\tau_1}. \quad C$$

увеличением количества задач алгоритма объем пересылок остается постоянным, длительность вычислений снижается. Следовательно, критерием эффективности по времени следует считать критерий δ .

3.1.2 Параллельные алгоритмы для явных разностных схем, двумерный случай.

Перейдем к рассмотрению двумерного уравнения теплопроводности

$$\frac{\partial U}{\partial t} - a \left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \right) = 0 \quad \text{и простейшей явной схемы для него}$$

$$\frac{U_{i,j}^{n+1} - U_{i,j}^n}{h_t} - a \left(\frac{U_{i-1,j}^n - 2U_{i,j}^n + U_{i+1,j}^n}{h_x^2} + \frac{U_{i,j-1}^n - 2U_{i,j}^n + U_{i,j+1}^n}{h_y^2} \right) = 0 \quad (3.2) \quad \text{на}$$

декартовой прямоугольной сетке $\omega^2 = \{(t_k, x_i, y_j), t_k = kh_t, k=0, 1, \dots, K=T/h_t, x_i = ih_x, i=1, \dots, I=L_x/h_x+1, y_j = jh_y, j=1, \dots, J=L_y/h_y+1\}$, наложенной на область D , имеющую форму прямоугольника со сторонами L_x, L_y .

Выбор декомпозиции, в конечном счете, определяется выбором модели передачи сообщений. При пересылке небольшого количества информации пакетами большого объема (пакетная модель) предпочтительнее одномерная

декомпозиция области. Если же полагать длительность пересылки линейно зависящей от объема пересылаемых данных (линейная модель), то для квадратной сеточной области выгоднее двумерная декомпозиция. Здесь пересылаются только границы подобластей, поэтому принимаем пакетную модель. Пытливый читатель самостоятельно повторит вывод ускорений алгоритмов для линейной модели передачи сообщений.

Рассмотрим одномерную декомпозицию сеточной области ω^2 (рис. 3.3).

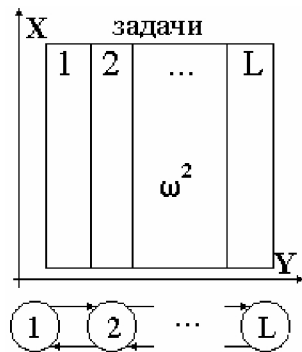


Рис. 3.3 Одномерная декомпозиция сеточной области ω^2 с разбиением по оси Y и топология коммуникаций "линия", соответствующая такому разбиению.

Используем вычислительную схему предыдущего алгоритма для построения нового параллельного алгоритма. На первом шаге вместо формулы (3.1) применим (3.2). Тогда для вычисления сеточной функции на новом временном слое потребуются значения

$$U_{i-1,j}^n, U_{i,j}^n, U_{i+1,j}^n, U_{i,j-1}^n, U_{i,j+1}^n.$$

for i=1 to I do begin

алгоритм для одномерной области для строки i по формуле (3.2);

end.

Все задачи алгоритма одновременно вычисляют значения сеточных функций на новом временном слое при $i=\text{const}$; $1 \leq j \leq J$. Если $I > J$, то выгоднее разбиение вдоль оси X, в этом случае задачи алгоритма будут вычислять сеточные функции при $j=\text{const}$; $1 \leq i \leq I$.

Ускорение такого алгоритма составит $S = \frac{I \times J \times \tau_1}{\gamma_{J,L}(I) \times I \times \tau_1 + 4\tau_n I}$, где $4I$ -

количество принятых и отправленных задач l ($2 \leq l \leq L-1$) пакетов. Ускорение представленного алгоритма равно ускорению предыдущего.

При модификации предложенного алгоритма воспользуемся тем, что пакеты отправляются по каналам почти пустыми, в них находится всего по одному числу. Сократим количество пакетов за счет их более рационального наполнения. Если I мало настолько, что I значений сеточной функции поместятся в один пакет, то целесообразно использовать другую вычислительную схему.

for $i=1$ to I do begin

первый шаг алгоритма для одномерной области для строки i по формуле (3.2);

end.

Затем задача l ($1 \leq l \leq L-1$) передает одним пакетом соседу справа вектор значений $U_{i,j}^{n+1}$, в котором $j = r_2^1$, $1 \leq i \leq I$. Задача l ($2 \leq l \leq L$) принимает от соседа слева вектор с индексами $j = r_1^1 - 1$, $1 \leq i \leq I$. Аналогично производится передача векторов от правых соседей - левым. При этом длительность пересылок алгоритма снижается в I раз.

$$\text{Ускорение} \quad S = \frac{I \times J \times \tau_1}{\gamma_{J,L}(I) \times I \times \tau_1 + 4\tau_n} = \frac{J \times \tau_1}{\gamma_{J,L}(I) \times \tau_1 + \frac{4}{I} \tau_n} \xrightarrow{L \rightarrow J} \frac{J}{1 + \frac{4}{I} k_1}$$

существенно возрастает. В случае, когда I значений сеточной функции в один пакет не помещаются, изложенную модификацию алгоритма также следует применять. Длительность пересылок по сравнению с первым вариантом алгоритма сократится в I/I_n раз, где I_n - максимальное количество значений сеточной функции, помещающихся в один пакет.

Двумерная декомпозиция сеточной области представлена на рис. 3.4.

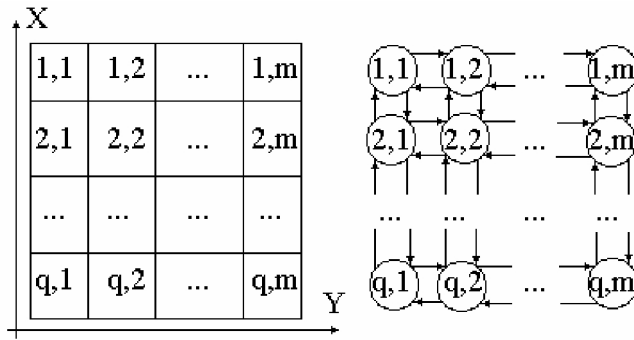


Рис. 3.4 Двумерная декомпозиция ω^2 и топология коммуникаций "решетка", соответствующая такому разбиению. Задачи алгоритма нумеруются двумя индексами l_x, l_y , как блоки матрицы. $L=m \times q$.

Настоящий алгоритм, как и предыдущий состоит из двух шагов: вычислительного и связанного с пересылкой сообщений. На первом шаге задача l_x, l_y ($1 \leq l_x \leq q$, $1 \leq l_y \leq m$) находит значения $U_{i,j}^{n+1}$ для $r_1^{l_x} \leq l_x \leq r_2^{l_x}$, $r_1^{l_y} \leq l_y \leq r_2^{l_y}$. Второй шаг характеризуется пересылкой векторов сначала по направлению X (сверху вниз и снизу вверх), затем по направлению Y (слева направо и справа налево).

Наибольший объем данных пересылают задачи l_x, l_y ($2 \leq l_x \leq q-1$, $2 \leq l_y \leq m-1$). Каждая из них передаст и примет от четырех соседей 8 пакетов, если $\max\{\gamma_{l,q}(1), \gamma_{l,m}(1)\} \leq I_n$. Ускорение такого алгоритма

$$S = \frac{I \times J \times \tau_1}{\gamma_{I,q}(I) \times \gamma_{J,m}(I) \times \tau_1 + 8\tau_n}.$$

При равном числе задач, эффективность по

времени предыдущего алгоритма вдвое превысит эффективность по времени эффективности настоящего. Исключение составляет случай, когда $\min\{m, q\} = 2$. Например, рассматриваемый четырехзадачный алгоритм по эффективности δ не уступает предыдущему алгоритму, топология его коммуникаций соответствует четырехпроцессорному кольцу. Интересно, что фактическое количество значений сеточных функций, пересылаемое одной задачей такого алгоритма составит всего половину от аналогичной характеристики четырехзадачного предыдущего алгоритма, но в рамках пакетной модели длительность этих пересылок окажется одинаковой.

В общем случае использование алгоритма с линейным разбиением предпочтительнее. Кроме меньшего объема передач он отличается простотой реализации, довольствуясь менее громоздкой сетевой конфигурацией.

Вкратце коснемся декомпозиции круговой области D . Сравним на прямоугольной сетке два варианта одномерного разбиения и вариант двумерного (рис. 3.5).

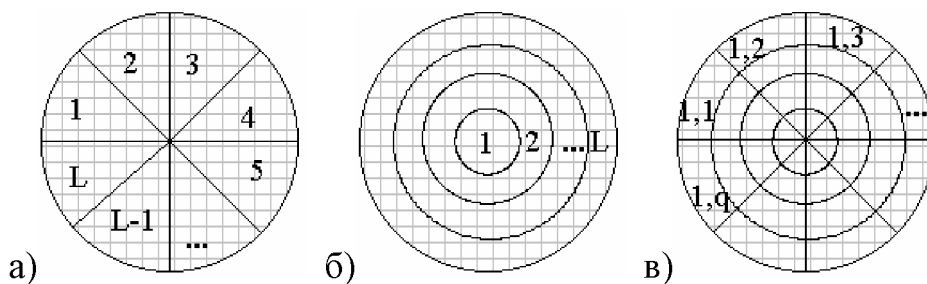


Рис. 3.5 Декомпозиции круговой области D с наложенной на нее прямоугольной сеткой. а) - одномерная декомпозиция с разбиением по углу φ ; б) - одномерная декомпозиция с разбиением по радиусу ρ ; в) - двумерная декомпозиция с разбиениями по углу и радиусу.

Одномерное разбиение по углу (рис. 3.5а) удачно тем, что при равномерном шаге по φ нарезаются сектора с одинаковым количеством узлов сетки. Запись алгоритма с такой декомпозицией отличается простотой: все задачи производят одинаковое количество арифметических операций, обмениваясь равным количеством значений сеточных функций. К недостатку алгоритма следует отнести сложную топологию коммуникаций, связанную с вычислениями в центральных узлах сетки. Для нахождения сеточных функций в таких узлах могут потребоваться обмены не только с соседями по и против часовой стрелки, но и с другими задачами (рис. 3.6). При реализации такого алгоритма нельзя ограничиться использованием сетевой конфигурации "кольцо".

Этого недостатка лишен алгоритм, использующий одномерную декомпозицию с разбиением по радиусу (рис. 3.5б). Все центральные узлы сеточной области принадлежат первой задаче такого алгоритма. Правда при

равномерном разбиении по ρ количество узлов сетки для каждой задачи окажется различным. Положение спасает неравномерное разбиение по радиусу.

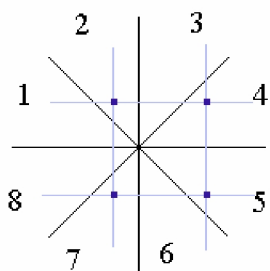


Рис. 3.6 Особенность вычисления функций в центральных узлах сетки при одномерной декомпозиции по углу. Для задачи 7, кроме каналов, связывающих ее с задачами 8 и 6 необходима установка дополнительных каналов до задач 2 и 5.

Количество пересылаемых значений сеточной функции варьируется в зависимости от номера задачи, но в принятой нами пакетной модели при известных ограничениях это несущественно.

Алгоритм с двумерной декомпозицией на рис. 3.5в характеризуется недостатками предыдущих алгоритмами, без их достоинств. К тому же его эффективность по времени окажется вдвое меньше.

При решении задач математической физики на круговую область обычно накладывают криволинейную сетку (рис. 3.7).

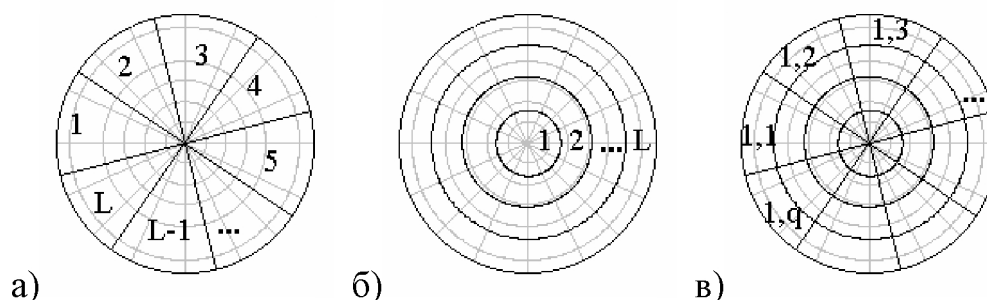


Рис. 3.7 Декомпозиции круговой области D с наложенной на нее криволинейной сеткой. а) - одномерная декомпозиция с разбиением по углу φ ; б) - одномерная декомпозиция с разбиением по радиусу ρ ; в) - двумерная декомпозиция с разбиениями по углу и радиусу.

При одномерном разбиении по углу (рис.3.7а), проблема центральной точки становится особенно острой. Коммуникации такого алгоритма глобальны. Появляются простые задачи, связанные с необходимостью решать сеточное уравнение для центрального узла в одной определенной задаче.

Коммуникации алгоритма с разбиением по радиусу (рис.3.7а) останутся локальными. Отпадет необходимость в неравномерном разбиении по r для равномерной сеточной области. Границы подобластей разных задач содержат одинаковое количество узлов. Если по r сетка гуще, чем по φ , то для линейной модели пересылки сообщений, разбиение по радиусу позволит сократить время обменов по сравнению с разбиением по углу.

Двумерное разбиение на рис. 3.7в будет полезным исключительно в рамках линейной модели, превосходя (на сетках с одинаковой дискретизацией по φ и по r) по эффективности оба варианта одномерного разбиения.

3.2. Параллельные алгоритмы решения трехдиагональных сеточных уравнений неявных разностных схем.

3.2.1. Алгоритм для одномерной сеточной области.

При разностном решении дифференциального уравнения на одномерной сеточной области ω_1 следует говорить об одной СЛАУ трехдиагонального вида (применяются неявные разностные схемы)

Функциональная декомпозиция задается спецификой метода встречных прогонок [6], в котором прогоночные коэффициенты (α , β и η , ζ) вычисляются с двух сторон по направлению к центру матрицы A . В последовательном алгоритме этот прием используется, когда желают ограничиться нахождением не всего вектора x , а лишь его части. Так как вычисления двух пар коэффициентов прогонки независимы (информационно, логически и конкурентно), то их можно находить параллельно.

Произведем разбиение одномерной сеточной области ω_1 на ω_1^1 и ω_1^2 (рис. 3.8).

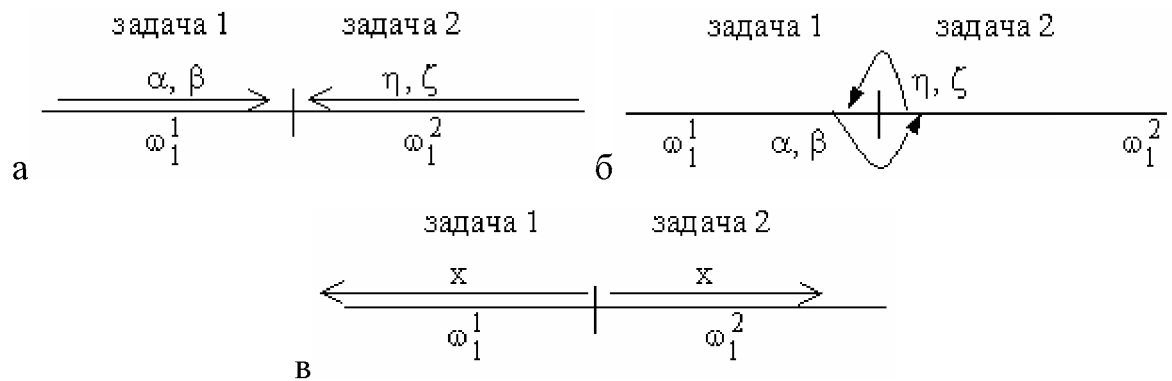


Рис. 3.8 Этапы вычислений по двухзадачному параллельному алгоритму, реализующему метод встречных прогонок на одномерной сеточной области. а - прямой ход прогонок; б - обмен данными между задачами; в - обратный ход прогонок. Верхний индекс при ω есть номер задачи, нижний – размерность области.

Вычисления разделим на три шага, соответствующие прямому (рис. 3.8а), обратному ходу (рис. 3.8в) встречных прогонок и обмену данными между задачами (рис. 3.8б).

В течение прямого хода прогонок первая задача находит прогоночные коэффициенты α, β , вторая задача - η, ζ . Далее, первая задача передает второй пару чисел α, β , необходимых для запуска обратного хода прогонки во второй задаче, которая в это время производит аналогичную передачу, необходимую первой задаче. На третьем шаге обе задачи одновременно реализуют обратный ход прогонок, находя решение СЛАУ. Ускорение такого алгоритма можно оценить как:

$$S = \frac{C_1 M \tau_a}{\frac{1}{2} C_1 M \tau_a + 2 \tau_k},$$

где M - размерность СЛАУ, $C_1 M \tau_a$ - длительность расчета по последовательному алгоритму, если τ_a - время производства одной арифметической операции с плавающей точкой, $\tau_k = \tau_y + \tau_n$ есть длительность одной коммуникации между задачами алгоритма (τ_y - время установки на

обмен, τ_n - время передачи или приема (считаем их равными) одного пакета по сети). При пакетной модели передачи данных время передачи разного объема информации остается константой до тех пор, пока объем передаваемого массива не превысит размер пакета. Здесь это условие соблюдается (передается всего пара чисел), в дальнейшем также будем полагать его верным. К сожалению, этот подход нельзя развить для большего количества задач (алгоритм не масштабируемый), если сеточная область ω одномерна (здесь он приведен исключительно для иллюстрации дальнейшего изложения).

Состоятельность приведенного двухзадачного алгоритма подтверждается результатами вычислительных экспериментов, представленных на рис. 3.9.

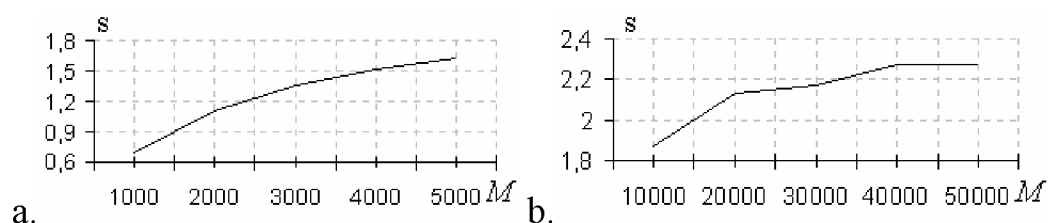


Рис. 3.9 Зависимость ускорения параллельных вычислительных процессов (ПВП), порожденных алгоритмом из пункта 3.2.1, от размерности сеточной области, для разных диапазонов M .

Все вычислительные эксперименты проводились на кластере Самарского научного центра РАН (СНЦ РАН), состоящем из двухпроцессорных ЭМВ Pentium III (600 МГц), связанных сетью Fast Ethernet.

Для исследуемого алгоритма ускорение, согласно закону Амдала, не должно превышать 2. Однако, как следует из рис. 2б, при $M=20000$, $S>2$. Для изучения этого эффекта обратимся к табл. 3.1. Таблица 3.1.

Зависимость объема оперативной памяти, занимаемой последовательным вычислительным процессом, от размерности ω_1 .

	размерность сеточной области ($M/1000$)									
	1	2	3	4	5	10	20	30	40	50
объем занимаемой памяти (кБ)	23	47	70,1	93,5	117,9	234,7	468,5	703,2	937,5	1171,7

Начиная с размерности $M=2000$ (именно с нее ускорение превышает 2), объем памяти, занимаемый последовательным вычислительным процессом, превышает кэш процессора (256кБ). Следовательно, последовательный и параллельный процессы оперируют областями памяти, характеризующимися разными скоростями доступа к данным. Область памяти, занимаемая последовательным процессом при $M=50000$, выходит за размеры кэша на 915,87 кБ, а для параллельного вычислительного процесса (ПВП) этот показатель равен 329,93 кБ и сравним с объемом кэша. Логично предположить, что как только объем памяти, занимаемой вычислительным процессом (не важно каким именно: последовательным или параллельным), превысит кэш на несколько порядков, ускорение опять подчинится закону Амдала. Действительно, при $M=2000000$ и занимаемой памяти 45,78 мБ, $S=1,9969$.

3.2.2. Алгоритм для двумерной сеточной области, линейное разбиение.

Для двумерной области ω_2 ограничение на масштабируемость снимается /7/. Применительно к ней будем говорить о прогонках по строкам (продольных прогонках) и столбцам (поперечных прогонках) сеточной области, как это принято в методах расщепления и переменных направлений.

Если алгоритм состоит из двух задач, то модификации, по сравнению с предыдущим алгоритмом, не требуется, каждая СЛАУ в продольном направлении решается одновременно двумя задачами. Вычисления в поперечном направлении являются независимыми и не сопровождаются коммуникациями. Пусть размер сеточной области ω_2 - $N \times M$ узлов (линейное разбиение производится по M), тогда ускорение такого алгоритма составит

$$S_2^L = \frac{2C_1MN\tau_a}{C_1MN\tau_a + 2N\tau_k}.$$

Числитель есть сумма длительности продольных $NC_1M\tau_a$ и поперечных $MC_1N\tau_a$ прогонок. Первое слагаемое знаменателя также сформировано как

сумма длительностей продольных $NC_1M/2\tau_a$ и поперечных $M/2C_1N\tau_a$ прогонок, реализуемых одной задачей алгоритма на половине сеточной области. Длительность коммуникаций линейно зависит от N - количества узлов сеточной области, располагающихся на границе подобластей. Двухзадачный алгоритм, основанный на методе встречных прогонок, свободен от задержек вычислений, связанных с ожиданиями.

Для алгоритма из четырех задач производится следующее линейное разбиение сеточной области (рис. 3.10).

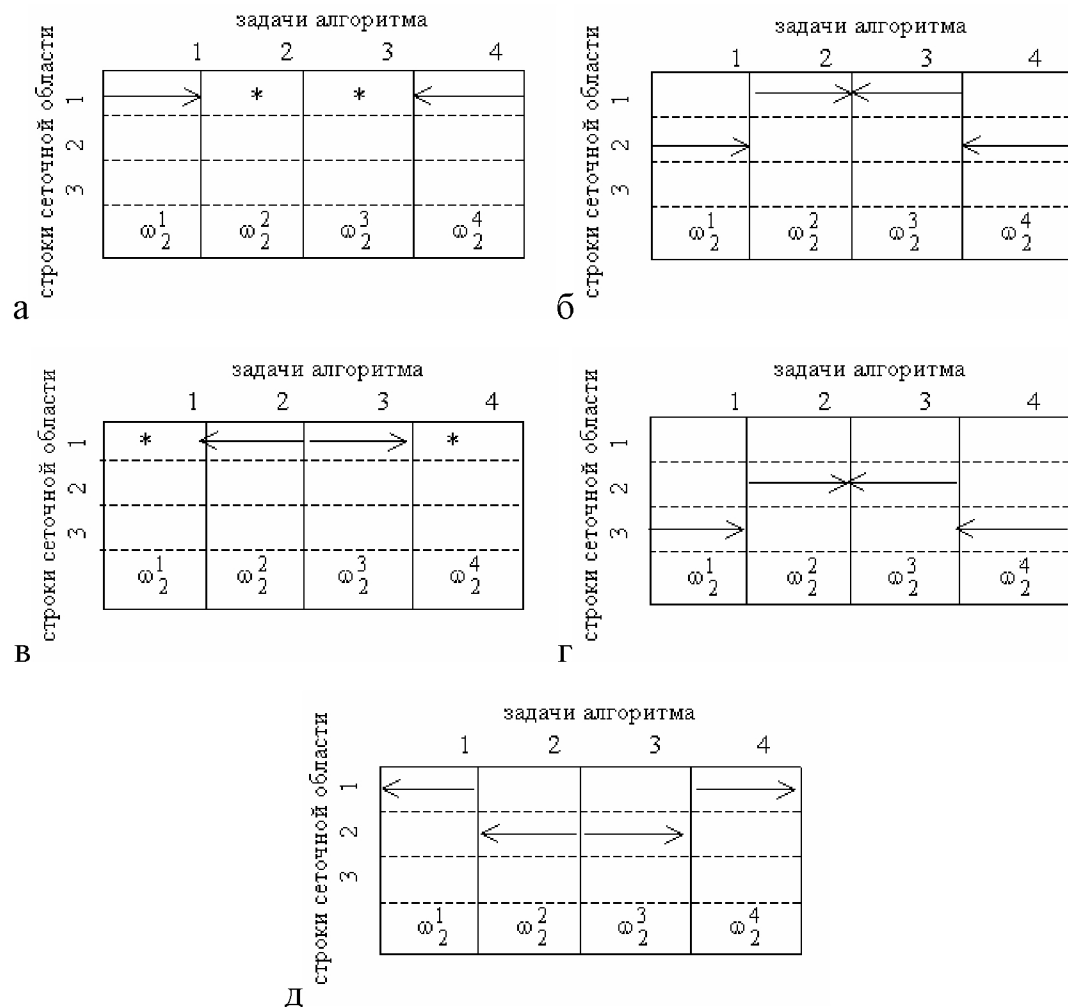


Рис. 3.10 Этапы вычислений по четырехзадачному параллельному алгоритму из пункта 2: а,б,г - прямые ходы прогонок; в,д - обратные ходы.

Задачи, помеченные символом "*", простаивают.

На рис. 3.10 представлены начальные шаги вычислений по алгоритму (его первый этап). Задачи 1, 4 (рис. 3.10а) начинают прямой ход прогонок для

строки 1 и передают прогоночные коэффициенты задачам 2, 3, которые на первом шаге простаивают.

Далее не будем специально говорить об обмене данными между задачами, подразумевая, что перед прямым ходом они принимают прогоночные коэффициенты, после прямого хода передают их. Перед обратным ходом принимают значение сеточной функции, после - передают его. Первая и последняя задачи начинают прямой ход без принятия прогоночных коэффициентов, а задачи под номерами $p/2$ и $p/2+1$ (p - общее число задач, в данном алгоритме $p=4$) начинают обратный ход принятием прогоночных коэффициентов (друг от друга).

На втором шаге (рис. 3.10б) задачи 2, 3 продолжают прямой ход для строки 1, задачи 1, 4 начинают прямой ход для строки 2. Третий шаг (рис. 4.5в) характеризуется простоем задач 1, 4 и началом обратного хода прогонок для строки 1 задачами 2, 3. Четвертый шаг (рис. 3.10г) - прямой ход, задачи 1, 4 осуществляют его для строки 3, задачи 2, 3 для строки 2. Пятый (рис. 3.10д) - обратный ход, задачи 1, 4 производят его для строки 1, задачи 2, 3 - для строки 2. Далее прямой и обратный ходы чередуются (второй этап алгоритма), пока задачи 2, 3 не произведут прямой ход прогонок для последней строки. На следующем этапе (заключительном) в течение первого шага задачи 1, 4 будут простаивать, задачи 2, 3 начнут обратный ход. Вторым шагом характеризуется простоем задач 2, 3 в то время, как задачи 1, 4 завершат обратный ход прогонок для последней строки сеточной области.

Ускорение данного алгоритма оценивается величиной

$$S_4^L = \frac{2C_1MN\tau_a}{\frac{1}{2}C_1MN\tau_a + 4N\tau_k + C_1\frac{M}{4}\tau_a},$$

где последнее слагаемое знаменателя есть время ожидания в разработанном алгоритме, связанное с простоями задач на начальной и заключительной стадиях вычислений.

Распространим данный подход на произвольное число задач p . На начальном этапе такого алгоритма в течение первых $p/2$ шагов производится только прямой ход прогонок. Задача l (пусть $l \leq p/2$) простаивает $l-1$ шаг, затем производит прямой ход прогонки для строк $1, 2, \dots, p/2-1$. После этого происходит чередование прямого и обратного хода, причем задача l простаивает $p/2-l$ шагов, связанных с обратным ходом. Далее продолжается чередование без простоев. Если в течение прямого хода задача l производит ход прогонки для строки i , то задача l производит прямой ход для строки $i-l+1$. Когда первая задача начинает обратный ход для строки i , задача l начинает обратный ход для $i-l+1$. Последние шаги алгоритма опять связаны с простоями, когда во время прямого хода простаивают задачи (для $l \leq p/2$) с меньшими номерами, во время обратного – с большими номерами. Аналогично проводятся рассуждения для $l > p/2$.

Ускорение и эффективность такого алгоритма составят:

$$S_p^L = \frac{2C_1MN\tau_a}{\frac{2C_1MN\tau_a}{p} + 4N\tau_k + \frac{p-2}{2p}C_1M\tau_a + (p-4)\tau_k} \text{ и}$$

$$E_p^L = \frac{2C_1MN\tau_a}{2C_1MN\tau_a + 4pN\tau_k + \frac{p-2}{2}C_1M\tau_a + p(p-4)\tau_k}.$$

Простои любой задачи связаны не только с ожиданием при производстве арифметических вычислений, но и с ожиданием коммуникаций между другими задачами алгоритма. Суммарное время простоев любой задачи одинаково.

В таблице 3.2 и на рис. 3.11 представлены результаты вычислительных экспериментов, подтверждающие работоспособность алгоритма на квадратной сеточной области.

Таблица 3.2

Зависимость ускорения вычислительного процесса от числа процессоров
и размерности квадратной сеточной области.

размерность сеточной области (N)	число процессоров (p)			
	2	4	6	8
1000	1,005392	0,944551	0,535323	0,520246
2000	1,085413	1,385219	1,790679	1,239702
5000	1,509048	2,208691	2,856274	2,791851
10000	1,680401	3,459153	4,353058	4,215099
15000	1,828832	3,811183	5,600946	6,691857

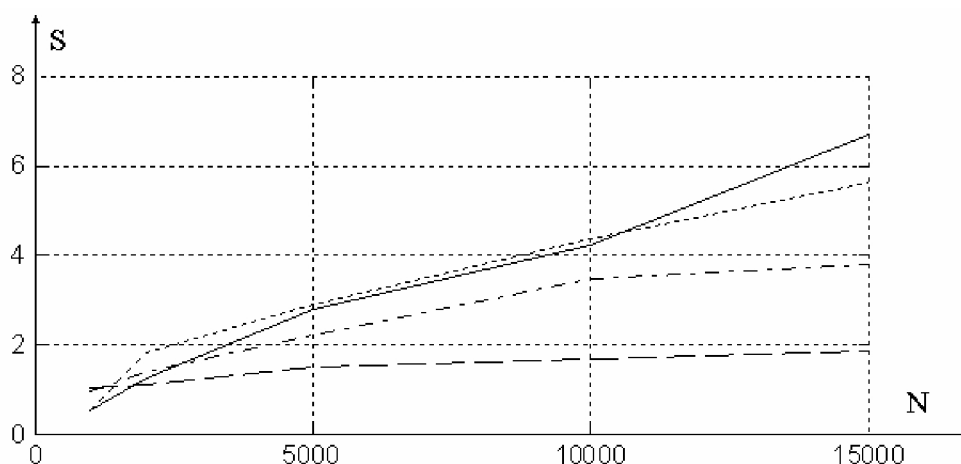


Рис. 3.11 Зависимости ускорений параллельных вычислительных процессов от размерности вычислительной области при реализации на двух (штриховая кривая), четырех (штрих-пунктир), шести (двойной пунктир) и восьми (сплошная кривая) процессорной вычислительной системе.

При увеличении размерности вычислительной области ускорение всех исследуемых параллельных вычислительных процессов возрастает до величин, близких к предельным по закону Амдала (рис. 3.11). Действительно, длительность коммуникаций при решении СЛАУ, соответствующей одной строке сеточной области, остается постоянной, а размерность такой СЛАУ увеличивается. Следовательно, увеличивается и длительность исполнения распараллеливаемых арифметических операций.

С другой стороны, решение систем малых размерностей ($N=1000$) вовсе не сопровождается ускорением вычислений по сравнению с

последовательным вычислительным процессом (таблица 3.2). Причем чем больше процессоров задействовано в реализации таких вычислений, тем меньшего ускорения удастся добиться. Наблюдаемый эффект связан с ростом коммуникационных издержек при увеличении числа процессоров. Для размерности $N=1000$ длительность арифметических операций на одном процессоре совпадает с длительностью коммуникаций между двумя процессорами. В итоге, ускорение двухзадачного вычислительного процесса составляет единицу (таблица 3.2). Для большего количества процессоров длительность коммуникаций увеличивается, время арифметических операций сокращается, в итоге ускорение снижается.

Повышению ускорения параллельных алгоритмов метода встречных прогонок с линейной декомпозицией сеточной области будет способствовать такая организация вычислений, при которой процессоры обмениваются не парой прогоночных коэффициентов и одним значением сеточной функции, а векторами, составленными из соответствующих параметров. Для этого можно организовывать коммуникации не сразу после прямого или обратного хода по одной строке (столбцу) сеточной области, а после накопления пересылаемых данных, завершив прямой (обратный) ходы для некоторого количества строк (столбцов) - блоков.

Однако такой прием ведет к увеличению объема памяти для хранения прогоночных коэффициентов, с которой оперирует вычислительный процесс, что может, с учетом иерархичности структуры кэш и оперативной памяти, негативно сказаться на скорости вычислений. Для разных типов вычислительных систем применение предложенного варианта может по-разному повлиять на ускорение, поэтому в настоящей работе рассматривается лишь численный вариант выбора параметров модифицированного алгоритма. При анализе ускорения параллельных алгоритмов сложно аналитически учесть все особенности конкретной вычислительной системы (работа с памятью, сетью и т.д.).

Для блочного варианта объём простоев, связанных с вычислениями, станет больше, зато уменьшится объём простоев, связанных с коммуникациями. Этот вариант особенно выгодно использовать в тех случаях, когда большая часть времени уходит на коммуникации, а не на арифметические операции – то есть при малых размерностях сеточных областей.

В соответствии с последней рекомендацией примем размерность сеточной области $N=1000$. В результате серии вычислительных экспериментов подберем такое количество строк в блоке (вектор которого и передается по сети между процессорами), при котором ускорение окажется максимальным.

В таблице 3.3 и на рис. 3.12 представлены результаты вычислительных экспериментов, подтверждающие работоспособность предложенной блочной модификации алгоритма.

Таблица 3.3а

Характеристики блочных процессов при вычислениях по сеточной области размерности $N=1000$ узлов. Зависимость ускорения от размера блока и числа процессоров.

ширина блока n (в строках)	число процессоров (p)		
	4	6	8
20	0,945781	0,641238	0,712346
40	1,000246	0,556889	0,664579
60	0,995633	0,934515	0,99785
80	1,512378	1,25488	1,23456
100	1,14567	1,512379	1,542346
120	1,207892	1,215954	1,523469

Таблица 3.3b

Характеристики блочных процессов при вычислениях по сеточной области размерности $N=1000$ узлов. Зависимость отношения ускорений блочного/неблочного вариантов от размера блока и числа процессоров.

ширина блока n (в строках)	число процессоров (p)		
	4	6	8
20	1,001303	1,197852	1,369248
40	1,058965	1,040285	1,277432
60	1,054081	1,745702	1,918034
80	1,601161	2,344154	2,37303
100	1,212926	2,825170	2,964645
120	1,278801	2,271439	2,928361

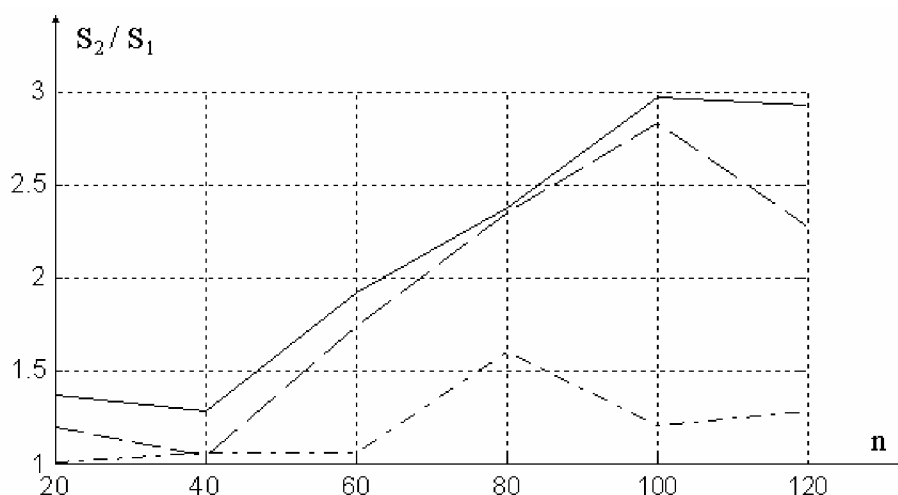


Рис. 3.12 Зависимость отношения ускорений блочного/неблочного вариантов (S_2/S_1) от размера блока (n) и числа процессоров (для $p=8$ сплошная кривая, для $p=6$ штриховая кривая, для $p=4$ штрих-пунктир) при $N=1000$ для линейного разбиения сеточной области.

Действительно, применение блочной модификации позволяет до трех раз (в случае восьми процессоров) повысить ускорение параллельного вычислительного процесса.

3.2.3. Алгоритм для двумерной сеточной области, циклическое разбиение.

Осуществим двумерное разбиение сеточной области ω_2 для построения параллельного алгоритма метода встречных прогонок. Очевидный вариант такого разбиения на четыре подобласти и этапы алгоритма представлены на рис. 3.13, его ускорение

$$S_4^C = \frac{2C_1MN\tau_a}{\frac{1}{2}C_1MN\tau_a + (N+M)\tau_k}.$$

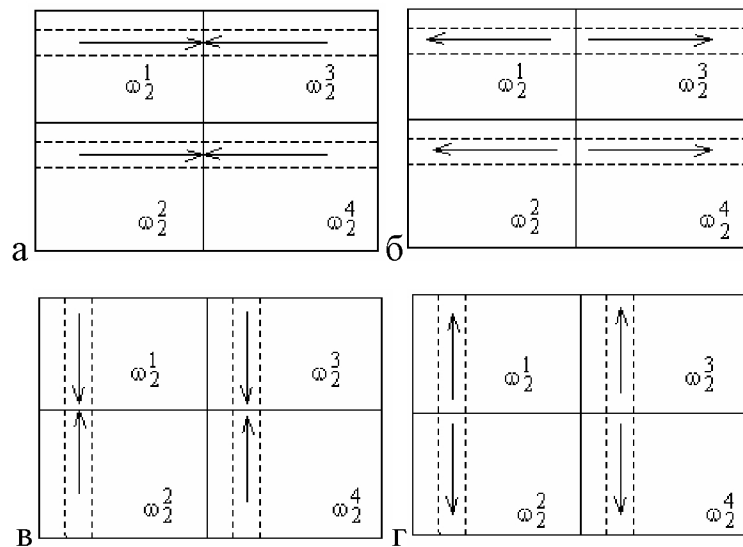


Рис. 3.13 Этапы вычислений по четырехзадачному параллельному алгоритму из пункта 3.2.3: а - прямой ход прогонок по строкам, б - обратный ход прогонок по строкам, в - прямой ход по столбцам, г - обратный ход по столбцам.

На первом этапе алгоритма (рис. 3.13а, 3.13б) задачи 1, 3 и 2, 4 производят встречные прогонки для строк $1, \dots, N/2$ и $N/2+1, \dots, N$, обмениваясь между собой прогоночными коэффициентами (1-ый с 3-им, 2-ой с 4-ым). На втором, заключительном, этапе (рис. 3.13в, 3.13г) задачи 1, 2 и 3, 4 производят встречные прогонки для столбцов $1, \dots, M/2$ и $M/2+1, \dots, M$, обмениваясь между собой прогоночными коэффициентами (1-ый со 2-ым, 3-й с 4-ым). Отсутствие простоев связано с тем, что при таком разбиении все задачи имеют возможность одновременно начать и завершить действия по алгоритму.

Циклическое разбиение области и действия по алгоритму из восьми задач иллюстрируются на рис. 3.14.

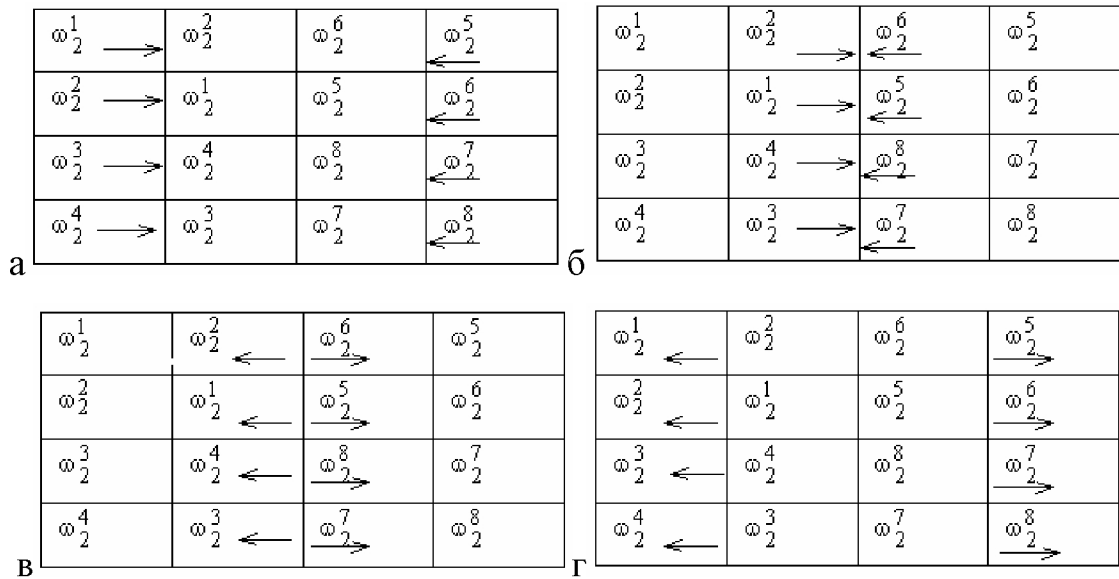


Рис. 3.14 Этапы вычислений по восьмизадачному параллельному алгоритму из пункта 3.2.3: а,б - прямой ход прогонок по строкам, в,г - обратный ход прогонок по строкам.

Особенность распределения ω_2 между задачами состоит в принадлежности одной задаче двух прямоугольных подобластей (циклическое разбиение). Их взаимное расположение в шахматном порядке позволяет распараллеливать вычисления прогонок как по строкам, так и по столбцам области, избегая простоев.

На первом этапе алгоритма (рис. 3.14а) производится прямой ход прогонок по строкам в левой и правой четвертях ω_2 . Так, задачи 1, 5 работают со строками 1,...,N/4; задачи 2, 6 - со строками N/4+1,...,N/2; задачи 3, 7 - со строками N/2+1,...,3N/4 и задачи 4, 8 - со строками 3N/4+1,...,N. Причем первые четыре задачи оперируют частями своих строк с 1 по M/4 столбец, последние четыре задачи - с 3M/4+1 по M столбец. В конце этапа пары задач 1, 2; 3, 4; 5, 6 и 7, 8 обмениваются прогоночными коэффициентами.

На втором этапе (рис. 3.14б) прямой ход прогонок по строкам заканчивается, вычисления из левой четверти переносятся на одну подобласть вправо, из правой четверти - на одну подобласть влево. Задачи 1,

5 работают со строками $N/4+1, \dots, N/2$; задачи 2, 6 со строками $1, \dots, N/4$; задачи 3, 7 - со строками $3N/4+1, \dots, N$, и задачи 4, 8 со строками $N/2+1, \dots, 3N/4$. Причем первые четыре задачи оперируют частями своих строк с $M/4+1$ по $M/2$ столбец, последние четыре задачи - с $M/2+1$ по $3M/4$ столбец. В конце этапа пары задачи 1, 5; 2, 6; 3, 7 и 4, 8 обмениваются прогоночными коэффициентами.

Третий и четвертый этапы алгоритма (рис. 3.14г,д) - обратный ход прогонок по строкам. Аналогично производятся прогонки по столбцам. Ускорение такого алгоритма

$$S_8^C = \frac{2C_1MN\tau_a}{\frac{C_1MN\tau_a}{4} + \frac{3}{2}(N+M)\tau_k}.$$

Циклическое разбиение ω_2 при $p=2^q$ и $q>3$ производится аналогично разбиению на рис.3.14. Каждой задаче будут принадлежать $p/4$ подобластей ω_2 и соседствовать $\log_2 p$ других задач. При этом части сеточной области, принадлежащие задачам с 1-ой по $p/4$ -ую, располагаются в четверти с координатами $1 \leq i \leq N/2, 1 \leq j \leq M/2$; с $p/4+1$ -ой по $p/2$ -ую - в четверти с координатами $N/2+1 \leq i \leq N, 1 \leq j \leq M/2$; с $p/2+1$ -ой по $3p/4$ -ую - в четверти с координатами $1 \leq i \leq N/2, M/2+1 \leq j \leq M$; с $3p/4+1$ -ой по p -ую - в четверти с координатами $N/2+1 \leq i \leq N, M/2+1 \leq j \leq M$. При выбранном “шахматном” порядке расположения никакие две подобласти, принадлежащие одной задаче, не содержат одну и ту же строку или столбец сеточной области.

При этом ускорение и эффективность разработанного алгоритма составят

$$S_p^C = \frac{2C_1MN\tau_a}{\frac{2C_1MN\tau_a}{p} + 2\left(1 - \frac{2}{p}\right)(N+M)\tau_k} \text{ и}$$

$$E_p^C = \frac{2C_1MN\tau_a}{2C_1MN\tau_a + 2p\left(1 - \frac{2}{p}\right)(N+M)\tau_k}.$$

Начиная постановку вычислительных экспериментов для сформулированного алгоритма, исследуем поведение реализации сформулированного в 3.2.3 алгоритма для сеточных областей малых размерностей (таблица 3.4, рис. 3.15).

Таблица 3.4

Ускорения параллельных вычислительных процессов по алгоритму с циклическим разбиением сеточной области для N от 100 до 1000.

P	N									
	100	200	300	400	500	600	700	800	900	1000
4	0,64	0,72	0,72	0,70	2,67	2,88	2,89	3,04	2,99	2,99
8	0,14	0,28	0,34	1,67	2,73	2,98	3,35	3,95	5,10	5,14

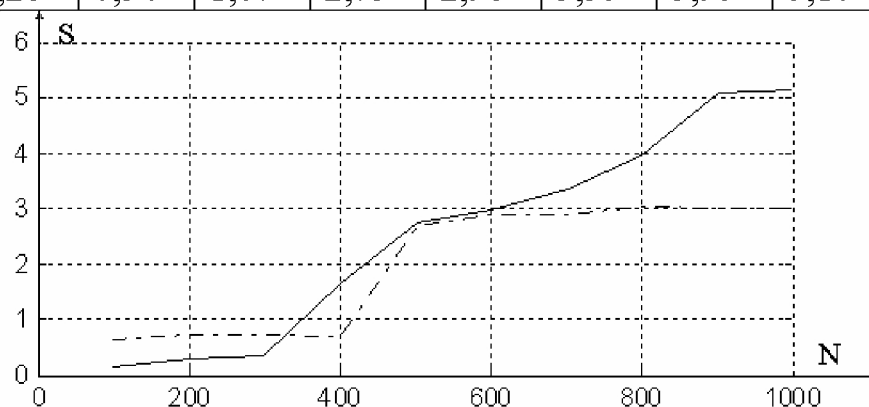


Рис. 3.15 Зависимости ускорений параллельных вычислительных процессов от размерности сеточной области (в диапазоне 100-1000), непрерывная кривая соответствует восьми процессорам, штрих пунктирная - четырем.

Сравнивая результаты на рис. 3.12 и 3.15 для N=1000 отметим большую эффективность алгоритма с циклическим разбиением сеточной области. В последней серии экспериментов ускорение достигается для областей, размерностью 300-400 узлов, в то время как при линейном разбиении для области размерностью 1000 узлов ни один вычислительный процесс не характеризуется ускорением. Причиной тому служит отсутствие простоев и меньший объем коммуникаций алгоритма с циклическим разбиением сеточной области.

Продолжая постановку вычислительных экспериментов, исследуем поведение алгоритма на сеточных областях с большей размерностью (таблица 3.5, рис. 3.16).

Таблица 3.5

Ускорения параллельных вычислительных процессов по алгоритму с циклическим разбиением сеточной области для N от 1000 до 10000.

P	N									
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
4	2,99	2,93	2,90	2,97	3,10	3,09	3,11	3,18	3,66	3,69
8	5,14	5,23	6,67	6,74	6,74	6,75	6,81	6,87	7,12	7,66

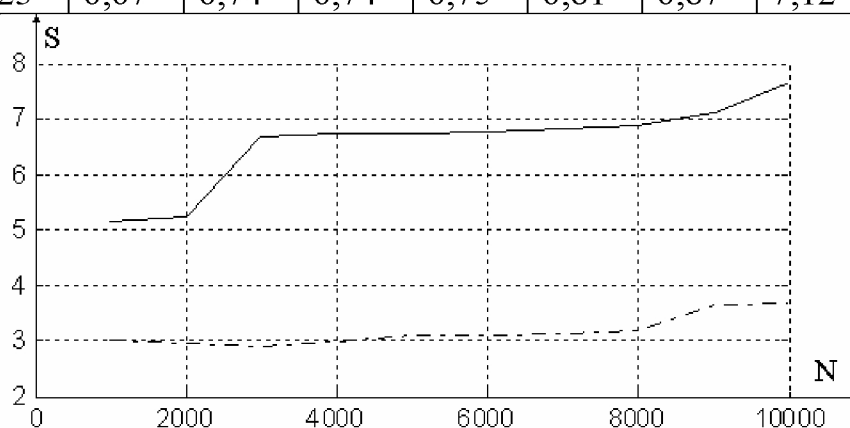


Рис. 3.16 Зависимости ускорений параллельных вычислительных процессов от размерности сеточной области (в диапазоне 1000-10000), непрерывная кривая соответствует восьми процессорам, штрих пунктирная - четырем.

Почти достигнув предельных показателей, значения ускорений незначительно изменяются при увеличении размерностей сеточных областей. При N=1000 только для $p=8$ параллельный вычислительный процесс метода встречных прогонок с циклической декомпозицией существенно превосходит по ускорению процесс метода с линейной декомпозицией. Требование соответствия закону Амдала уравнивает различные варианты метода встречных прогонок на густых сеточных областях.

Проведем исследования эффективности блочного варианта метода, аналогичные предложенным в 3.2.2. В таблицах 3.6 -3.9 представлены ускорения для блочной модификации алгоритма встречных прогонок с

циклическим разбиением сеточной области, при размерах блока 20, 40, 60 и 80 соответственно.

Таблица 3.6а

Ускорение вычислительного процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 20 строк области.

P	N									
	100	200	300	400	500	600	700	800	900	1000
4	0,842	0,923	0,944	1,013	3,329	3,736	3,591	3,793	3,790	3,791
8	0,306	0,608	0,782	2,033	3,014	3,778	3,793	3,985	5,523	5,526

Таблица 3.6b

Отношение ускорений блочного/неблочного вариантов для процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 20 строк области.

P	N									
	100	200	300	400	500	600	700	800	900	1000
4	1,32	1,28	1,31	1,45	1,25	1,3	1,24	1,25	1,27	1,27
8	2,19	2,17	2,3	1,2	1,1	1,27	1,13	1,01	1,09	1,02

Таблица 3.7а

Ускорение вычислительного процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 40 строк области.

P	N									
	100	200	300	400	500	600	700	800	900	1000
4	0,946	1,013	1,013	1,020	3,380	3,780	3,749	3,796	3,797	3,797
8	0,381	0,580	0,819	2,126	3,056	3,796	3,809	3,994	5,560	5,530

Таблица 3.7b

Отношение ускорений блочного/неблочного вариантов для процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 40 строк области.

p	N									
	100	200	300	400	500	600	700	800	900	1000
4	1,48	1,41	1,41	1,46	1,27	1,31	1,3	1,25	1,27	1,27
8	2,72	2,07	2,41	1,27	1,12	1,27	1,14	1,01	1,09	1,08

Таблица 3.8a

Ускорение вычислительного процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 60 строк области.

p	N									
	100	200	300	400	500	600	700	800	900	1000
4	1,065	1,197	1,249	1,250	3,543	3,871	3,783	3,798	3,801	3,813
8	0,639	0,692	0,829	2,193	3,099	3,842	3,843	4,111	5,737	5,890

Таблица 3.8b

Отношение ускорений блочного/неблочного вариантов для процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 60 строк области.

p	N									
	100	200	300	400	500	600	700	800	900	1000
4	1,66	1,66	1,74	1,79	1,33	1,34	1,31	1,25	1,27	1,28
8	4,56	2,47	2,44	1,31	1,1	1,3	1,15	1,04	1,13	1,15

Таблица 3.9a

Ускорение вычислительного процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 80 строк области.

p	N									
	100	200	300	400	500	600	700	800	900	1000
4	1,162	1,296	1,320	1,273	3,601	3,892	3,789	3,877	3,862	3,895
8	0,767	0,760	0,831	2,265	3,143	3,846	3,907	4,141	5,783	4

Таблица 3.9b

Отношение ускорений блочного/неблочного вариантов для процесса по блочной модификации алгоритма встречных прогонок с циклическим разбиением сеточной области, размер блока 80 строк области.

p	N									
	100	200	300	400	500	600	700	800	900	1000
4	1,82	1,8	1,83	1,82	1,35	1,35	1,31	1,28	1,29	1,3
8	5,48	2,71	2,44	1,36	1,15	1,29	1,17	1,05	1,13	1,15

На рисунках 3.18 представлено сравнение блочного и не блочного вариантов алгоритмов для $N=1000$.

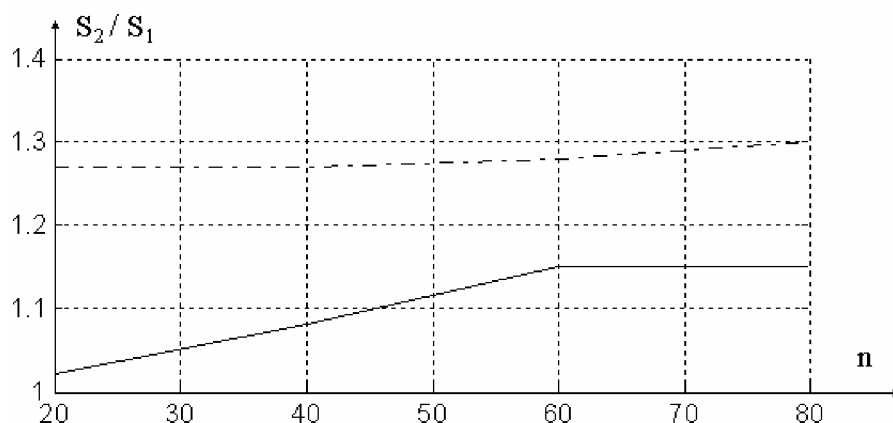


Рис. 3.18 Зависимость отношения ускорений блочного/неблочного вариантов (S_2/S_1) от размера блока (n) и числа процессоров (для $p=8$ - сплошная кривая, для $p=4$ - штрих-пунктир) при $N=1000$ для циклического разбиения сеточной области.

Так же, как и в случае линейного разбиения сеточной области (рис. 3.12), для варианта с циклическим разбиением (рис. 3.18) блочная модификация позволяет существенно улучшить результаты вычислительных экспериментов. С увеличением размерности сеточной области разница между блочным и неблочным вариантами сокращается. Это связано с увеличением объема памяти для хранения прогоночных коэффициентов одной строки сеточной области. Для определенной размерности указанный объем займет всю кэш память, в связи с чем разбиение области на блоки теряет смысл.

Заключение

В настоящем пособии автор представил наиболее распространенные методы построения параллельных алгоритмов, проиллюстрировав их примерами из теории разностных схем. Читатель понимает, что не существует универсальных методов, удачно подходящих для всех случаев из вычислительной практики.

Например, можно разработать параллельный алгоритм вычисления степенной функции x^n , имеющий топологию коммуникаций "дерево" и состоящий из $\log_2 n$ шагов для n задач. Тогда, реализуя на каждом шаге одно умножение, можно свести n умножений последовательного алгоритма к $\log_2 n$ умножениям параллельного.

Подойдем к задаче с другой стороны, воспользовавшись известным соотношением $\sum_{i=1}^n \frac{r_i}{x - r_i} = \frac{n}{x^n - 1}$, где r_i - корни уравнения $x^n = 1$. Тогда,

предварительно вычислив корни (они не зависят от значения x), построим другой параллельный алгоритм. На первом шаге найдем $a_i = x - r_i$, что одно сложение в каждой задаче. Второй шаг - $b_i = r_i / a_i$, одно деление. Далее, с помощью дерева вычислим $f = \sum_{i=1}^n b_i$ за $\log_2 n$ сложений. На четвертом шаге

вычислим $c = n / f$ (одно деление в первой задаче). Последний шаг - нахождение результата: $x^n = c + 1$ (одно сложение). В итоге, новый алгоритм содержит $\log_2 n + 2$ сложений и 2 деления. Длительность вычислений по нему сократится в t_x / t_+ раз по сравнению с первым алгоритмом.

Наиболее оптимальные параллельные алгоритмы строятся не по последовательным алгоритмам, для них разрабатываются специальные параллельные численные методы. И здесь читателю уже не помогут рекомендации из книжек, лишь собственная интуиция ("метод научного тыка") и настойчивость ("метод пристального взглядывания") поведут его к желанной цели.

Литература

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления// СПб.: БХВ-Петербург, 2002.-608с.
2. Вальковский В.А., Котов В.Е., Марчук А.Г., Миренков Н.Н. Элементы параллельного программирования .- М.: Радио и связь, 1983.-239 с
3. Foster J. Design and building parallel programs (<http://www.mcs.anl.gov/~itf/dbpp/>)
4. Деммель Дж. Вычислительная линейная алгебра. Теория и приложение// пер. с англ. –М.:Мир, 2001.-430 с.
5. Ортега Д. Введение в параллельные и векторные методы решения линейных систем . - М.: Мир , 1991.-364 с.
6. Самарский А.А., Николаев Е.С. Методы решения сеточных уравнений М.: Наука. - 1978.- 561 с.
7. Головашкин Д.Л. Параллельные алгоритмы решения сеточных уравнений трехдиагонального вида, основанные на методе встречных прогонок// Математическое моделирование, 2005. – Том 17, № 11.- С.118-128.