Серия
Суперкомпьютерное
Образование

Координационный совет
Системы научно-образовательных центров суперкомпьютерных технологий

Председатель Координационного совета

В. А. Садовничий,
ректор МГУ имени М. В. Ломоносова,
академик

Заместители председателя совета

Е. И. Моисеев,
декан факультета вычислительной математики и кибернетики
МГУ имени М. В. Ломоносова,
академик

А. В. Тихонравов,
директор Научно-исследовательского вычислительного центра
МГУ имени М. В. Ломоносова,
профессор

Члены совета

В. Н. Васильев, ректор национального исследовательского Санкт-Петербургского государственного университета информационных технологий, механики и оптики, профессор; **Г. В. Майер**, ректор национального исследовательского Томского государственного университета, профессор; **Е. В. Чупрунов**, ректор национального исследовательского Нижегородского государственного университета, профессор; **А. Л. Шестаков**, ректор национального исследовательского Южно-Уральского государственного университета, профессор; **В. Н. Чубариков**, декан механико-математического факультета МГУ имени М. В. Ломоносова, профессор; **М. И. Панасюк**, директор Научно-исследовательского института ядерной физики МГУ имени М. В. Ломоносова, профессор; **Вл. В. Воеводин**, заместитель директора Научно-исследовательского вычислительного центра МГУ имени М. В. Ломоносова, исполнительный директор НОЦ «СКТ-Центр», член-корреспондент РАН.

Томский государственный университет

А. В. Старченко, В. Н. Берцун

Методы параллельных вычислений

Допущено УМС по математике и механике УМО по классическому университетскому образованию в качестве учебника для студентов высших учебных заведений, обучающихся по направлениям подготовки 010100 Математика, 010200 Математика и компьютерные науки

Издательство Томского университета
2013

УДК 519.6
ББК 22.18
С 77

Рецензенты:
кафедра прикладной математики и информатики Томского государственного университета систем управления и электроники;
доктор физико-математических наук
М. А. Т о л с т ы х

Старченко А.В., Берцун В.Н.
С 77 Методы параллельных вычислений: Учебник. – Томск: Изд-во Том. ун-та, 2013. – 223 с. (Серия «Суперкомпьютерное образование»)
ISBN 978–5–7511–2145–7

Представлены математические основы параллельных вычислений и методы решения задач вычислительной математики с использованием многопроцессорных вычислительных систем с распределенной памятью: вычисления по рекуррентным формулам, базовые алгоритмы линейной алгебры, прямые и итерационные методы решения линейных систем уравнений, сплайны, вычисление определенных и кратных интегралов, быстрое преобразование Фурье, метод Монте-Карло, численное решение обыкновенных дифференциальных уравнений и уравнений в частных производных.

Для научных сотрудников, аспирантов, студентов и преподавателей, изучающих или использующих высокопроизводительные вычислительные алгоритмы и ресурсы в научной и учебной работе.

Подготовка учебника выполнена в рамках реализации проекта комиссии Президента РФ по модернизации и технологическому развитию экономики России «Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения».

Ключевые слова: параллельные вычисления, методы вычислительной математики, многопроцессорные вычислительные системы с распределенной памятью

УДК 519.6
ББК 22.18

ISBN 978–5–7511–2145–7

Уважаемый читатель!

Вы держите в руках одну из книг серии «Суперкомпьютерное образование», выпущенную в рамках реализации проекта комиссии Президента РФ по модернизации и технологическому развитию экономики России «Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения». Инициатором издания выступил Суперкомпьютерный консорциум университетов России.

Серия включает более 20 учебников и учебных пособий, подготовленных ведущими отечественными специалистами в области суперкомпьютерных технологий. В книгах представлен ценный опыт преподавания суперкомпьютерных технологий в таких авторитетных вузах России, как МГУ, ННГУ, ТГУ, ЮУрГУ, СПбГУ ИТМО и многих других. При подготовке изданий были учтены рекомендации, сформулированные в Своде знаний и умений в области суперкомпьютерных технологий, подготовленном группой экспертов Суперкомпьютерного консорциума, а также международный опыт.

Современный уровень развития вычислительной техники и методов математического моделирования дает уникальную возможность для перевода промышленного производства и научных исследований на качественно новый этап. Эффективность такого перехода напрямую зависит от наличия достаточного числа высококвалифицированных специалистов. Данная серия книг предназначена для широкого круга студентов, аспирантов и специалистов, желающих изучить и практически использовать параллельные компьютерные системы для решения трудоемких вычислительных задач.

Издание серии «Суперкомпьютерное образование» наглядно демонстрирует тот вклад, который внесли участники Суперкомпьютерного консорциума университетов России в создание национальной системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий, а также их четкое понимание ответственности за подготовку высококвалифицированных специалистов и формирование прочного научного фундамента, столь необходимого для эффективного использования суперкомпьютерных технологий на практике.

Ректор Московского университета,
Президент Суперкомпьютерного консорциума
университетов России,
академик РАН *В.А. Садовничий*

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современная наука опирается на наблюдения, теорию, эксперимент и математическое моделирование. Именно математическое моделирование является чрезвычайно важным инструментом получения новых знаний там, где невозможно провести наблюдения или экспериментальные исследования из-за их чрезвычайной дороговизны или значительных временных затрат, а также по другим важным причинам. Математическая модель поведения рассматриваемого объекта, построенная на основе принятых теоретических положений, представляет собой замкнутую систему уравнений, решение которой позволяет получить некоторую информацию об исследуемом процессе или объекте. Математическое моделирование изучает поведение не самого объекта, а его математической модели, и по полученным результатам оценивается или прогнозируется реальное состояние объекта.

Для того, чтобы математическая модель правдоподобно описывала реальность, она должна включать большое количество параметров объекта, часто взаимосвязанных, что значительно усложняет математическую формулировку модели и делает в большинстве случаев невозможным поиск аналитического решения полученной системы уравнений. Привлечение вычислительной техники позволяет находить приближенное решение сложных научно-технических и социально-экономических задач, формализованных методами математического моделирования. Среди современных видов высокопроизводительной вычислительной техники важное место занимают суперкомпьютеры, имеющие рекордные показатели по числу выполняемых в секунду арифметических операций (*floating point operation per second ~ flops*) и объему памяти.

В настоящее время среди многочисленных областей применения суперкомпьютеров можно выделить: стратегические исследования, промышленность, финансы, управление, медиа-технологии, телекоммуникации, научные исследования и образование.

Стратегические исследования включают следующие направления: космические и ядерные исследования, окружающая среда, безопасность. К космическим исследованиям относят разработку и запуск спутников и ракет, изучение космоса, защиту Земли от космических объектов, интерпретацию спутниковых данных и др. К ядерным ис-

следованиям – обеспечение безопасности хранения ядерного резерва, поиск новых источников энергии. Исследование окружающей среды подразумевает анализ и предсказание изменений климата и погоды, изучение и прогноз землетрясений, исследование океана и мониторинг окружающей среды.

Следующая важная область применения суперкомпьютеров – промышленность. Сюда, главным образом, относятся: добывающая, автомобильная, авиационная, космическая, электронная, оборонная. В добывающей промышленности суперкомпьютеры применяют для разведки полезных ископаемых, проектирования бурения, моделирования резервуаров. В автомобильной промышленности – для сокращения сроков разработки новых автомобилей при помощи компьютерного моделирования: проведения прочностных и других инженерных расчётов, исследования аэродинамики. В авиационной промышленности суперкомпьютерные технологии используются для проектирования авиационных двигателей и других элементов самолёта. В космической промышленности проводятся детальные инженерные расчёты для создания новых ракетных двигателей, космических аппаратов и их систем безопасности.

В такой области применения суперкомпьютеров, как финансы, решают следующие задачи: управление сверхбольшими базами данных банков, финансовый консалтинг (планирование, прогнозирование, управление рисками и т. д.).

В управлении суперкомпьютеры необходимы для работы со сверхбольшими базами данных крупных компаний, государственного управления и планирования (компании Philip Morris, BOSCH, Tech Pacific Exports (Индия), Oracle, BLG Logistics (Германия), европейская сеть супермаркетов Carrefour).

Наиболее впечатляющей областью применения мощных вычислительных систем являются медиа-технологии: разработка цифровых видеотехнологий для создания художественных и анимационных фильмов, компьютерных игр. Одним из примеров может служить вычислительный кластер компании DreamWorks, который используется для создания мультфильмов.

В такой области, как телекоммуникации, суперкомпьютеры применяются для различных телекоммуникационных сервисов для сверхбольших баз данных абонентов (France Telecom, Telcom South

Africa, Telecom Italia, Digital China, Deutche Telecom, Vodaphone и др.).

Самой большой областью применения суперкомпьютеров в мире пока остаётся область научных исследований и образования. Здесь рассматриваются задачи из различных отраслей науки, например:

- биоинформатика: изучение генома человека и открытие новых способов лечения;
- молекулярное моделирование: биомолекулярная динамика и изучение белковых структур для создания новых лекарственных препаратов;
- нанотехнологии: синтез новых материалов;
- квантовая химия, молекулярная динамика, физика частиц, астрофизика, динамическая астрономия и др.

Сегодня суперкомпьютер представляет собой совокупность взаимосвязанных вычислительных узлов с гибридной архитектурой (несколько многоядерных центральных процессоров и одна или несколько плат с графическими процессорами). По способу организации взаимодействия между вычислительными узлами многопроцессорные вычислительные системы делятся на системы с общей оперативной памятью и распределенной. В большинстве своем современные суперкомпьютеры в целом представляют собой многопроцессорные высокопроизводительные вычислительные системы (МВС) с распределенной памятью. В них каждый вычислительный узел имеет свою локальную оперативную память, а доступ к данным, находящимся в оперативной памяти другого узла многопроцессорной системы, осуществляется с использованием передачи сообщений по высокоскоростной сети, связывающей вычислительные узлы. Высокая скорость проведения расчетов на системах с распределенной памятью обусловлена возможностью одновременного выполнения вычислений при решении одной конкретной задачи математического моделирования на нескольких (в лучшем случае на всех имеющихся) процессорных элементах вычислительной системы при минимуме затрат на межпроцессорную передачу данных. Теоретически, сколько в вычислениях используется процессоров или ядер МВС, во столько раз быстрее должна выполняться программа на суперкомпьютере. Однако добиться этого не всегда удается. Важным является правильный учет особенностей архитектуры МВС и топологии межпроцессорных связей, при этом требуется модификация последовательного алго-

ритма или программы для вычислительной системы с параллельной архитектурой.

Для получения максимальной производительности последовательной программы необходимо, чтобы она (программа) соответствовала архитектуре вычислительной машины. По этой причине программа, написанная в машинных кодах и учитывающая структуру машины, может превосходить по производительности рабочую программу, составленную самым совершенным компилятором.

При применении МВС с параллельной архитектурой еще более важной становится роль качества программной реализации алгоритма решения задачи на используемой вычислительной технике. Если для последовательных компьютеров разница между хорошо и плохо написанной программой может составлять более 2–3 раз по времени работы процессора, то для ЭВМ с параллельной архитектурой это отличие может достигать 5–10 раз. Хотя сейчас и имеется программное обеспечение для автоматизированного распараллеливания численных алгоритмов (НОРМА, DVM, V-Ray, Linda, Occam, HPF, Par4All, Polaris, Intel C++, SUIF Compilers и др.), тем не менее роль программиста-математика в построении эффективного высокопроизводительного способа решения задачи на суперкомпьютере еще долго будет оставаться решающей.

Производительность компьютерной программы – величина, обратная процессорному времени, затрачиваемому на выполнение программы. Поэтому когда говорится о повышении производительности программы, то имеется в виду минимизация времени, затрачиваемого на её выполнение. Оценивание производительности программы возможно и по другим параметрам.

Для параллельных вычислительных систем эта цель не есть синоним минимального числа операций. Может случиться так, что для повышения производительности параллельной программы придётся добавить какое-то количество вспомогательных вычислений. Поэтому производительность вычислительной системы измеряют количеством выполняемых операций с плавающей точкой за секунду (флопс), а производительность программы для расчетов следует оценивать по-другому, учитывая качество программирования, используемые компиляторы, степень соответствия полученной рабочей программы архитектуре компьютера, на котором эта программа выполняется.

Степень параллелизма компьютера – количество информации, которое может быть обработано одновременно. Наибольший параллелизм высокопроизводительной вычислительной системы достигается при обработке количества информации, кратного степени параллелизма компьютера.

Степень параллелизма алгоритма – число арифметических операций, которое можно выполнять независимо, т.е. параллельно.

Например, рассмотрим задачу сложения двух векторов размерности n. Операции сложения компонент векторов независимы и могут выполняться параллельно. Следовательно, степень параллелизма рассмотренного алгоритма равна n. Заметим, что степень параллелизма алгоритма не связана с параметрами компьютера, а является внутренней характеристикой алгоритма. Однако если степень параллелизма в алгоритме соответствует степени параллелизма компьютера, то можно написать хорошую высокопроизводительную программу для этого компьютера.

Рассмотрим другой пример – нахождение скалярного произведения двух векторов: $\left(\vec{a},\vec{b}\right)=\sum_{i=1}^{n} a_i b_i;\ \ \vec{a},\vec{b}\in\mathbb{R}^n$. Здесь операция покомпонентного умножения может выполняться параллельно, однако сложение полученных произведений по обычному последовательному алгоритму будет неэффективно для параллельных вычислений. Несколько исправить ситуацию может применение алгоритма сдваивания при суммировании $e_i = a_i \times b_i, i=\overline{1,n}$, когда одновременно суммируются соседние пары слагаемых, а затем их суммы (см. табл. 0.1).

Таблица 0.1

Алгоритм сдваивания ($n = 2^3 = 8$)

Этапы	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
1	e_1+e_2		e_3+e_4		e_5+e_6		e_7+e_8	
2	$e_1+e_2+e_3+e_4$				$e_5+e_6+e_7+e_8$			
3	$e_1+e_2+e_3+e_4+e_5+e_6+e_7+e_8$							

Если $n = 2^q$, то алгоритм сдваивания состоит из q этапов (тактов): на первом этапе выполняется $n/2$ сложений, на втором – $n/4$, …, на последнем – одно сложение. Таким образом, степень параллелизма алгоритма может оставаться постоянной на всех этапах, а может и меняться.

Средней степенью параллелизма численного алгоритма называется отношение общего числа арифметических операций к числу его этапов.

Для суммирования векторов число операций равно n, число этапов алгоритма – 1, средняя степень параллелизма равна n. Для скалярного произведения получаем соответственно $2n-1$, $\log_2 n+1$ и $(2n-1)/(\log_2 n+1)$. Так как степень параллелизма операции суммирования двух векторов совпадает со средней степенью параллелизма этого алгоритма, то он обладает «идеальным» параллелизмом. Для алгоритма нахождения скалярного произведения средняя степень параллелизма в $(\log_2 n+1)$ раз меньше идеальной степени параллелизма $(2n-1)$. Однако если при суммировании использовать последовательный алгоритм, то средняя степень параллелизма будет $(2n-1)/n$.

Поэтому главная задача разработчика параллельных программ – найти метод решения задачи, который отражает наилучшее соответствие между параллелизмом алгоритма и параллелизмом компьютера.

В качестве показателей оценки эффективности параллельной программы обычно рассматривают ускорение и эффективность.

Ускорение параллельной программы (алгоритма), получаемое при запуске программы на системе с p процессорами, – это отношение

$$S_p = \frac{T_1}{T_p},$$

где T_1 – время выполнения программы на одном процессоре; T_p – время выполнения программы на системе из p процессоров.

Эффективность использования параллельной программой ресурсов многопроцессорной вычислительной системы при решении задачи определяется соотношением

$$E_p = \frac{S_p}{p}.$$

При создании высокопроизводительных программ необходимо стремиться к тому, чтобы $S_p \to p$ и $E_p \to 1$.

Заметим, что наряду с замерами времени выполнения параллельных программ на многопроцессорных компьютерах можно также проводить предварительные расчеты времени выполнения алгоритма на основе анализа его сложности (количества выполняемых арифметических операций) для изучения перспективности выбранного или разработанного параллельного алгоритма. И хотя такие расчеты не в полной мере учитывают особенности реальной архитектуры многопроцессорных вычислительных систем, тем не менее они могут быть весьма полезны на начальном этапе разработки параллельного алгоритма при анализе его производительности и масштабируемости.

Проводя теоретические оценки для задачи нахождения суммы двух векторов, получим $T_1 \approx t_{add} \cdot n$; $T_p \approx \frac{n}{p} \cdot t_{add}$ и, следовательно, $S_p \approx p$. Здесь t_{add} – время выполнения операции сложения.

Для задачи определения скалярного произведения векторов с использованием последовательного алгоритма суммирования имеем:

$$T_1 \approx (t_{add} + t_{mult}) \cdot n;$$

$$T_p \approx \frac{n}{p} \cdot (t_{add} + t_{mult}) + p \cdot (t_{add} + t_{comm}); \quad S_p \approx \frac{p}{1 + \frac{p^2 \cdot \alpha}{n}}.$$

А при применении алгоритма сдваивания получим:

$$T_p \approx \frac{n}{p}(t_{add} + t_{mult}) + \log_2 p \cdot (t_{add} + t_{comm}); \quad S_p \approx \frac{p}{1 + \frac{p \cdot \log_2 p \cdot \alpha}{n}},$$

где $\alpha = \frac{t_{add} + t_{comm}}{t_{add} + t_{mult}} >> 1$, t_{add} – время, необходимое для сложения двух чисел с плавающей точкой, t_{mult} – время, необходимое для умноже-

ния двух чисел с плавающей точкой, t_{comm} – время, необходимое для передачи одного вещественного числа между процессорами. Здесь предполагается, что передача данных между любыми двумя процессорными элементами может происходить напрямую, т.е. МВС имеет топологию «полный граф», когда каждый процессорный элемент непосредственно в сети связан с любым другим процессорным элементом. Здесь и далее под процессорным элементом (ПЭ) будем понимать устройство, включающее микропроцессор, локальную память и некоторые вспомогательные схемы.

Из приведенного ниже рис. 0.1 видно, что даже при отсутствии затрат на межпроцессорную передачу данных каждый из рассмотренных алгоритмов вычисления скалярного произведения не обладает идеальным параллелизмом. При неизменном размере задачи (n=const) с ростом числа используемых процессоров темп роста $S_p(n, p)$ снижается, что можно объяснить увеличением доли коммуникационных затрат в общем времени выполнения параллельного алгоритма.

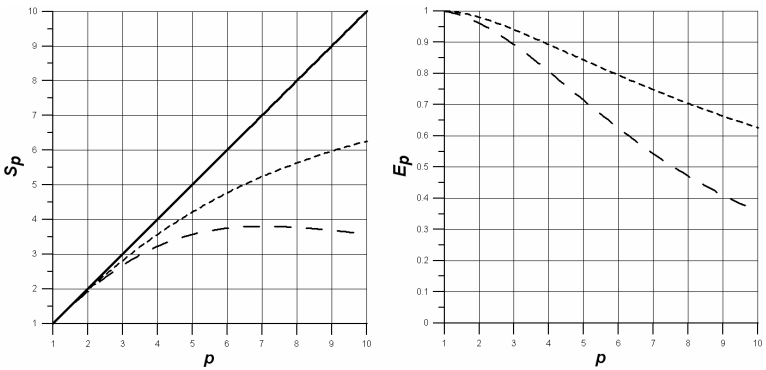


Рис. 0.1. Графики приближенной оценки ускорения и эффективности параллельного алгоритма скалярного произведения двух векторов (n = 1000) при использовании последовательного алгоритма суммирования (крупный штрих) и алгоритма сдваивания (мелкий штрих). α = 20

Также из рисунка следует, что применение алгоритма сдваивания повышает ускорение и эффективность алгоритма скалярного произведения. Анализируя полученные аналитические оценки ускорения

для различных способов организации суммирования результатов вычислений каждого ПЭ, можно отметить, что при неизменном размере задачи график ускорения $S_p = S_p(n, p)$ имеет максимум, в котором достигается наибольшее ускорение параллельного алгоритма и дальнейшее увеличение числа используемых процессоров становится нецелесообразным (точка насыщения параллельного алгоритма). Заметим, однако, что обе полученные выше формулы для ускорения параллельного алгоритма скалярного произведения в предельном случае при $n \to \infty$ показывают результаты, близкие к идеальному ($S_p = p$).

Рассмотрим обобщенную модель ускорения

$$S_p = \frac{T_1}{(\alpha_1 + \alpha_2 / k + \alpha_3 / p) \cdot T_1 + T_{comm}},$$

где α_1 – доля операций в параллельной программе, выполняемых последовательно (работает только один процессор, остальные $p-1$ простаивают);

α_2 – доля операций в параллельной программе, выполняемых со средней степенью параллелизма $k < p$;

α_3 – доля операций, выполняемых с максимальной степенью параллелизма p;

$\alpha_1 + \alpha_2 + \alpha_3 = 1;\ 0 \le \alpha_i \le 1, i = 1,2,3$;

T_1 – время исполнения одним процессором наиболее эффективной последовательной программной версии алгоритма;

T_{comm} – время, необходимое для подготовки данных (передачи их на задействованные процессорные элементы) для проведения параллельных вычислений на МВС.

Рассмотрим несколько специальных случаев.

Случай 1. $\alpha_1 = \alpha_2 = 0;\ \alpha_3 = 1,\ T_{comm} = 0$, следовательно, операции выполняются с максимальным параллелизмом $S_p = p$ и задержки при параллельном выполнении алгоритма отсутствуют.

Случай 2. $\alpha_1 = \alpha_3 = 0;\ \alpha_2 = 1,\ T_{comm} = 0$, и, следовательно, ускорение равно лишь средней степени параллелизма $\left(S_p = k < p\right)$.

Случай 3. $\alpha_2 = 0; T_{comm} = 0; \alpha_1 = \alpha; \alpha_3 = 1 - \alpha$, т.е. в задаче $(\alpha \cdot 100)\%$ от всех операций выполняется последовательно, а остальные $[(1-\alpha)\cdot 100]\%$ – с максимальным параллелизмом. В таком случае получаем закон Амдала (Amdahl's Law): $S_p = \dfrac{1}{\alpha + (1-\alpha)/p}$, согласно которому независимо от количества используемых процессоров и при игнорировании затрат на подготовку данных ускорение ограничено величиной, обратной доле последовательных операций алгоритма (программы), т.е. $S_p < \dfrac{1}{\alpha}$. Например, если в параллельной программе 10% операций выполняется последовательно, ее ускорение не может быть более 10 при любом числе используемых процессоров. Этот закон показывает, что в эффективном параллельном алгоритме величина α_1 должна быть минимальна. Суперкомпьютеры обычно используются для того, чтобы решать задачи большого размера, и в них, как правило, относительная доля последовательно выполняющихся операций уменьшается с ростом размера задачи. Причем часто темп уменьшения напрямую зависит от числа используемых при расчетах процессоров.

Случай 4. Время T_{comm} очень велико ($T_{comm} > T_1$), и может получиться, что $S_p < 1$ независимо от того, какие значения имеют $\alpha_1, \alpha_2, \alpha_3$. Это возможно, когда временные затраты на подготовку данных для проведения параллельных вычислений очень велики, и в этом случае целесообразно использовать однопроцессорную машину.

Таким образом, при разработке параллельного алгоритма решения сложной научно-технической задачи весьма полезным является предварительный анализ, который позволит прогнозировать эффективность алгоритма, установит его сильные и слабые места, что может быть использовано при выборе путей повышения эффективности создаваемой параллельной программы.

Этапы разработки параллельных программ

После того, как был проведен предварительный анализ и принято решение по построению параллельной программы, можно приступать к ее созданию. Заметим, что предлагаемая ниже схема разработ-

ки параллельных алгоритмов и программ (рис. 0.2) может использоваться и в процессе проведения предварительного анализа.

При разработке параллельных программ выделяют четыре этапа: декомпозиция, проектирование коммуникаций, укрупнение и планирование вычислений.

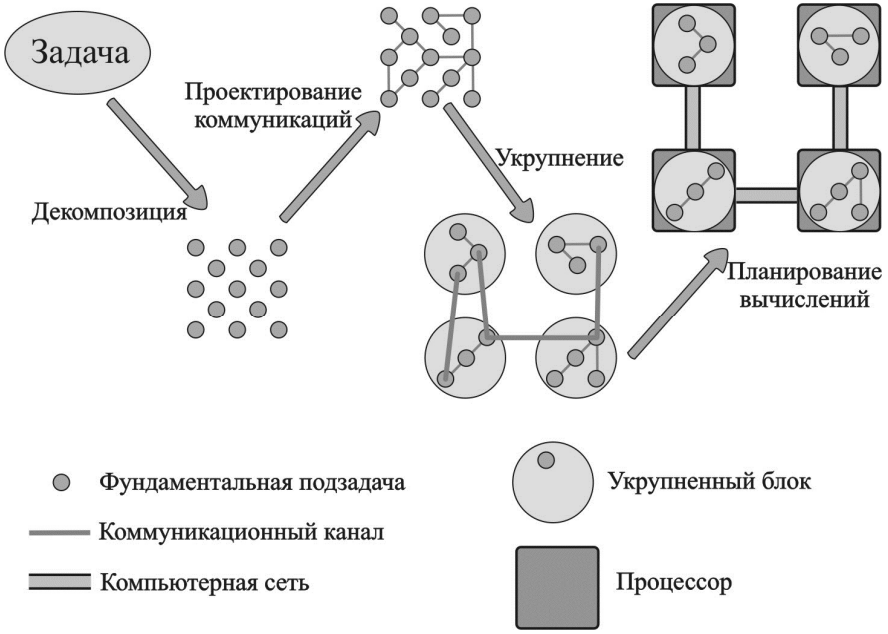


Рис. 0.2. Общая схема разработки параллельного алгоритма

Декомпозиция. На этом этапе задача анализируется, проводится оценка возможности ее распараллеливания. В случае перспективности создания эффективной параллельной программы производится разделение задачи на более мелкие части – фрагменты структур данных и фрагменты алгоритма (фундаментальные подзадачи). Такое разделение необходимо для повышения максимального параллелизма задачи. Количество фундаментальных подзадач должно быть, по крайней мере, на порядок больше планируемого числа используемых процессоров, а сами подзадачи в перспективе должны иметь примерно одинаковый размер. Пересечение выделенных фрагментов алгоритма должно быть сведено к минимуму, чтобы избежать дублиро-

вания вычислений (следствие – снижение эффективности параллельного алгоритма). Декомпозиция должна быть такой, чтобы при увеличении размера задачи увеличивалось бы количество фундаментальных подзадач, а не размер каждой подзадачи.

Размер фундаментальной подзадачи определяется степенью детализации распараллеливаемого алгоритма, которая характеризуется количеством операций в подзадаче. Если в алгоритме можно выделить фрагменты (подзадачи), в которых выполняется лишь несколько операций, то говорят о мелкозернистом параллелизме. Если фрагменты алгоритма (фундаментальные подзадачи) определяются на уровне процедур, в которых количество арифметических операций составляет порядка 10^3, то имеет место среднеблочный параллелизм. Крупнозернистый параллелизм алгоритма характеризуется возможностью выделения нескольких крупных задач на уровне отдельных программ, которые могут выполняться на компьютере с параллельной архитектурой независимо, что требует, как правило, специальной организации вычислений.

На этом этапе разработки программы для параллельных вычислений не учитываются особенности архитектуры многопроцессорной вычислительной системы.

Различают декомпозицию по данным и функциональную декомпозицию. В первом случае первоначально фрагментируются данные, с которыми связываются операции алгоритма их обработки, а во втором – сначала декомпозируется алгоритм обработки данных. Примерами для декомпозиции по данным могут служить одномерная, двумерная или трехмерная декомпозиция преобразуемых по определенным правилам элементов массива, имеющего три и более индексов. Одновременное вычисление по последовательным алгоритмам суммы, разности, скалярного и векторного произведений двух заданных векторов размером n – это пример функциональной декомпозиции.

Проектирование коммуникаций между подзадачами. На этом этапе устанавливаются коммуникации (связи) между фундаментальными подзадачами. Определяется коммуникационная модель передачи данных между подзадачами. Выбираются алгоритмы и методы коммуникаций. При этом необходимо учитывать типы коммуникаций. Так, наличие только локальных коммуникаций, когда каждая подзадача связана лишь с небольшим числом других подзадач, дает

преимущество при построении параллельной программы по сравнению с глобальными коммуникациями (каждая подзадача связана с большим числом других подзадач). Также проще строить параллельную программу, если тип коммуникаций структурированный (коммуникации образуют регулярную структуру, например с топологией «решетка»), статический (коммуникационная структура не меняется с течением времени), синхронный (отправитель и получатель полностью координируют передачу данных между собой). Однако использование динамических (коммуникационная структура, связывающая подзадачи, меняется с течением времени) и асинхронных (отправитель не контролирует получение данных, а получатель – их отправку) коммуникаций открывает новые возможности в создании эффективных параллельных программ.

При разработке коммуникационной модели передачи данных между фундаментальными подзадачами необходимо выполнять следующие требования:

- у каждой подзадачи количество коммуникаций (связей) с другими фрагментами алгоритма должно быть примерно одинаковым;
- объем одновременно передаваемых по коммуникациям данных должен быть одинаковым;
- предпочтение в использовании имеют локальные коммуникации, по которым передача данных осуществляется одновременно;
- по возможности передачу данных лучше совмещать с вычислениями;
- передача данных по коммуникациям не должна приводить к неодновременному выполнению фундаментальных подзадач.

На этом этапе не учитывается архитектура суперкомпьютера.

Укрупнение. Производится объединение подзадач в более крупные блоки для повышения эффективности создаваемого алгоритма – чтобы, например, снизить коммуникационные затраты или трудоемкость разработки параллельного алгоритма. Учитывается архитектура многопроцессорной системы, для которой разрабатывается параллельная программа. Количество укрупненных блоков фундаментальных подзадач должно соответствовать числу используемых процессоров (ядер). Первыми кандидатами для объединения в укрупненные блоки являются фундаментальные подзадачи, которые не могут выполняться одновременно и независимо. При укрупнении иногда удается сни-

зить коммуникационные затраты за счет замены передачи данных дублированием вычислений в различных подзадачах или блоках подзадач.

Укрупнение должно выполняться таким образом, чтобы сохранялись масштабируемость и высокая производительность параллельной программы. Говорят, что алгоритм или программа являются масштабируемыми, если эффективность $E_p(n, p)$ можно удерживать на постоянном ненулевом значении при одновременном увеличении числа используемых процессоров p и размера задачи n.

Планирование вычислений. На этом этапе осуществляется назначение укрупненных подзадач определенным процессорам в соответствии с требованиями снижения коммуникационных затрат и обеспечения параллелизма. Особое значение этот этап имеет при проведении вычислений на гетерогенных многопроцессорных системах, характеризующихся наличием различных по производительности вычислительных узлов и компонентов межпроцессорной сети. Стратегия размещения укрупненных блоков подзадач на процессорных элементах строится с учетом основного критерия – минимизации времени выполнения параллельной программы. Чаще всего используются стратегии «хозяин/работник», иерархические и децентрализованные схемы (рис. 0.3–0.5).

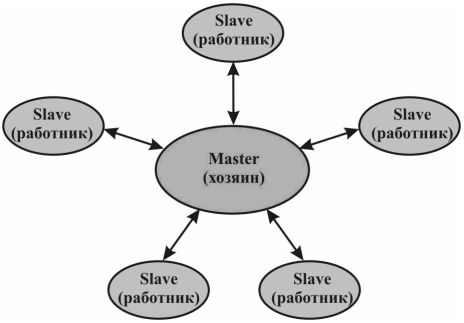


Рис. 0.3. Стратегия «хозяин/работник» размещения укрупненных блоков по шести процессорным элементам

В стратегии «хозяин/работник» главный блок (master) отвечает за непересекающееся распределение основной вычислительной нагрузки (укрупненных блоков подзадач) между процессорами-работниками (slave), контролирование ее выполнения и сбор расчетных данных для подготовки основного результата решения задачи (рис. 0.3).

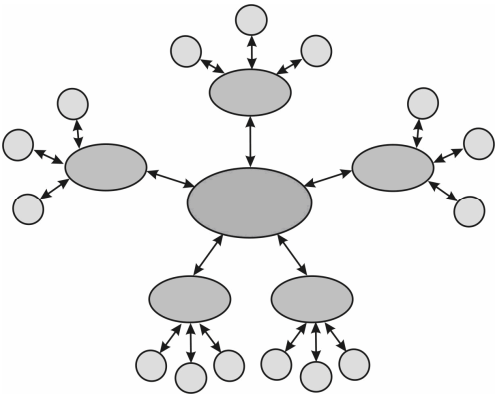


Рис. 0.4. Иерархическая схема «хозяин/работник» размещения укрупненных блоков по процессорным элементам

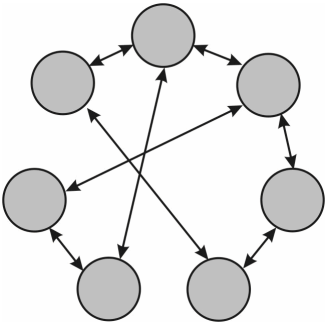


Рис. 0.5. Децентрализованная схема размещения укрупненных блоков по процессорным элементам

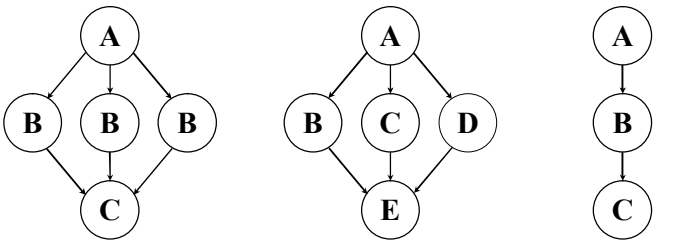
В иерархической схеме «хозяин/работник» объем вычислительной работы также разделен между процессорами-работниками, однако каждый из них выступает в качестве процессора-«хозяина» по от-

ношению к процессорам-«работникам», расположенным на более низкой ступени в этой иерархической схеме (рис. 0.4). Характер взаимодействия между уровнями иерархии подобен взаимодействию процессоров в стратегии «хозяин/работник». В децентрализованной схеме главный блок отсутствует, вычисления в блоках выполняются, придерживаясь определенной стратегии (рис. 0.5).

В рассмотренной выше схеме разработки параллельных алгоритмов и программ важную роль в идентификации параллелизма алгоритма играет анализ графа алгоритма, который может быть построен после выполнения этапов декомпозиции и проектирования коммуникаций.

Граф алгоритма представляет собой ориентированный граф, состоящий из вершин, соответствующих определенным фрагментам (фундаментальным подзадачам) алгоритма, и направленных дуг, отвечающих за передачу данных между ними (рис. 0.6). По дугам (коммуникациям) результаты, полученные при выполнении фрагментов алгоритма, передаются в качестве аргументов для продолжения алгоритма другим фрагментам. Стрелка из вершины A к вершине B графа алгоритма означает, что задача A должна быть выполнена перед задачей B. В этом случае имеет место зависимость задачи B от результата выполнения задачи A. Если задачи A и B не связаны между собой, то они независимы и могут выполняться одновременно (параллельно).

На рис. 0.6 представлены графы некоторых алгоритмов. Вершины представляют собой подзадачи или фрагменты алгоритмов. Буква внутри вершины определяет операцию алгоритма. Стрелки означают зависимость между подзадачами.



Алгоритм 1 Алгоритм 2 Алгоритм 3

Рис. 0.6. Параллелизм в графах алгоритмов

Граф Алгоритма 1 представляет параллелизм по данным. Три подзадачи с различными аргументами могут выполнять операцию B.

Граф Алгоритма 2 показывает функциональный параллелизм. Подзадачи, реализующие операции B, C, D, могут выполняться одновременно.

Граф Алгоритма 3 сответствует линейным зависимостям между операциями A, B, C, которые выполняются друг за другом.

1. РЕКУРРЕНТНЫЕ ФОРМУЛЫ. ВЫЧИСЛЕНИЕ ЧАСТИЧНЫХ СУММ

Рекуррентные вычисления – это последовательные вычисления, при которых значение рассматриваемого члена получаемой последовательности зависит от одного или нескольких предыдущих. В качестве примеров можно указать формулы для вычисления факториала, чисел Фибоначчи, метод прогонки. Рекуррентные вычисления составляют определённую проблему для параллельного компьютера.

1.1. Вычисление частичных сумм

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

$$S_k = \sum_{i=1}^{k} x_i, \quad 1 \le k \le n, \tag{1.1}$$

где n – количество суммируемых значений.

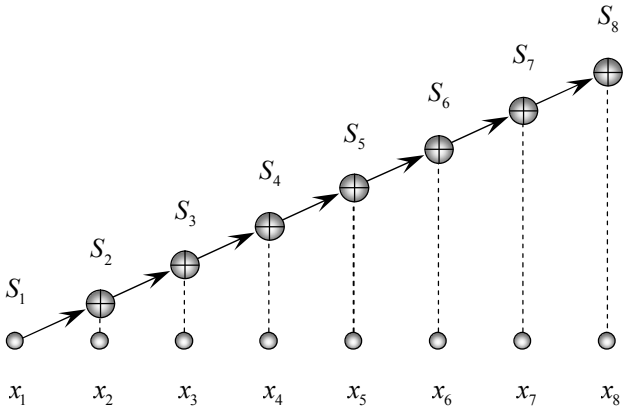


Рис.1.1. Схема последовательного алгоритма вычисления частных сумм ($n = 8$)

В последовательном алгоритме частные суммы можно вычислить весьма просто из рекуррентного соотношения

$$S_k = S_{k-1} + x_k; \quad k = \overline{1,n}; \quad S_0 = 0. \tag{1.2}$$

Соотношение между способом хранения данных, арифметическими операциями и временем можно изобразить на диаграмме маршрутизации (рис. 1.1). Последовательность вычислений перемещается снизу вверх. Операции, которые могут выполняться параллельно, показаны на одном и том же вертикальном уровне (такте). Ясно, что на каждом временном уровне может выполняться только одна операция (степень параллелизма равна единице), поэтому алгоритм последовательных сумм выполняется за $(n-1)$ тактов со степенью параллелизма 1. Рассмотренный алгоритм суммирования допускает только строго последовательное исполнение.

Параллельная реализация алгоритма суммирования становится возможной только при ином способе построения процесса вычисления, основанном на использовании ассоциативности операции сложения. Получаемый новый вариант суммирования (каскадная схема) состоит в следующем:

- набор из $p = n$ $(n = 2^q)$ процессорных элементов (ПЭ$_i$, i=1,…,n) сначала загружается данными $\{x_i\}, i = 1,...,n$, которые должны суммироваться, и на каждом ПЭ$_i$ выполняется $S_i = x_i$;

- на первом этапе копия S_i из ОЗУ каждого ПЭ$_i$ передается соседнему справа ПЭ$_{i+1}$ и складывается с S_{i+1};

- на следующем этапе процесс повторяется, но уже со сдвигом на две позиции;

….

- на k-м этапе сдвиг осуществляется на 2^k позиции;

….

После q $(q = \log_2 n)$-го этапа S_i каждого ПЭ содержит требуемые частные суммы.

Таблица 1.1

Каскадная схема суммирования $(n = 4, q = 2)$

	ПЭ$_1$	ПЭ$_2$	ПЭ$_3$	ПЭ$_4$
Номер этапа	S_1	S_2	S_3	S_4
k=0	1	2	3	4
k=1	1	3	5	7
k=2	1	3	6	10

В таблице 1.1 представлен пример выполнения каскадной схемы суммирования для последовательности $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4$.

Метод каскадных сумм требует $\log_2 n$ сложений со степенью параллелизма n, если для обеспечения однородности вычислительной схемы на ПЭ, для которых нет данных слева, отправлять нулевые значения (рис. 1.2). Каждый такт алгоритма каскадных сумм имеет максимально возможную степень параллелизма n. Общее число скалярных арифметических операций за счет «дублирования вычислений» увеличилось с $n-1$ до $n\log_2 n$, а время выполнения задачи уменьшилось. Таким образом, увеличение степени параллелизма от 1 до n в каскадной схеме суммирования на каждом этапе происходит за счёт существенного увеличения числа скалярных арифметических операций. Например, при n = 64 – в 6 раз.

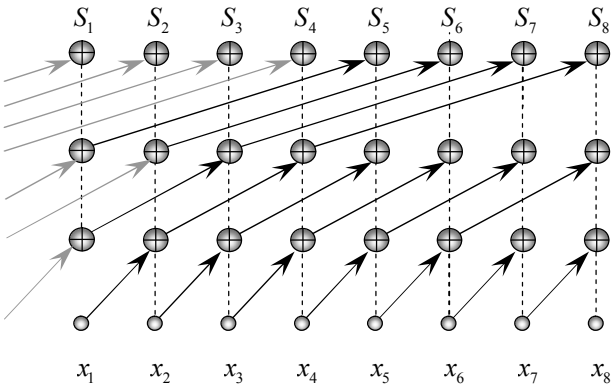


Рис.1.2. Каскадная схема нахождения частных сумм

Подсчитаем для рассмотренного алгоритма нахождения частных сумм показатели ускорения и эффективности, не учитывая затраты на пересылку данных:

$$S_p = \frac{T_1}{T_p} \approx \frac{n-1}{\log_2 n}; \quad (p = n); \quad E_p = \frac{n-1}{n\log_2 n} \approx \frac{1}{\log_2 n}.$$

Видно, что эффективность алгоритма уменьшается при увеличении числа суммируемых значений.

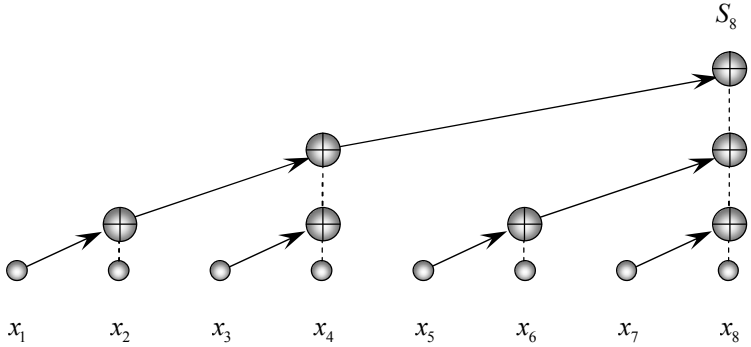


Рис. 1.3. Каскадная схема суммирования (алгоритм сдваивания) для вычисления суммы элементов последовательности

Алгоритм каскадного суммирования частных сумм значительно упрощается в случае, если требуется найти только значение общей суммы S_n (рис. 1.3):

- на первом этапе копия $S_i = x_i$ из оперативной памяти ПЭ с нечетными индексами копируется направо с последующим суммированием;

- далее на k-м этапе ($k = 1,2,...,(\log_2 n - 1)$) копия полученной суммы с ПЭ$_{\mu}$ ($\mu \bmod 2^{k+1}=2^k$) передается на ПЭ$_{\mu+2^k}$, где она добавляется к имеющейся в памяти сумме.

После завершения процесса на последнем ПЭ получается требуемый результат. Такой алгоритм называется алгоритмом сдваивания.

Как видно из рис. 1.3, вычисления, направленные на нахождение общей суммы, формируют двоичное дерево, где число операций на каждом вычислительном этапе (параллелизм алгоритма) уменьшается вдвое. Количество вычислительных этапов оказывается равным $\log_2 n$, а общее количество операций суммирования

$$\frac{n}{2}+\frac{n}{4}+...+2+1=\frac{1-2^{\log_2 n}}{1-2}=n-1$$

совпадает с количеством операций последовательного алгоритма суммирования. Кроме того, на каждом этапе проводится одно сложение со степенью параллелизма $n \cdot 2^{-k}$, $k = 1, 2, ..., \log_2 n$. Для эффективности и ускорения алгоритма сдваивания получим:

$$S_p = \frac{T_1}{T_p} \approx \frac{n-1}{\log_2 n}, E_p = \frac{T_1}{pT_p} \approx \frac{n-1}{p \cdot \log_2 n} = \frac{n-1}{(n/2)\log_2 n},$$

где $p = n/2$ – необходимое для выполнения каскадной схемы количество ПЭ. Заметим, что $E_p \to 0$ при $n \to \infty$.

Рассмотрим параллельную схему суммирования элементов последовательности с асимптотически ненулевой эффективностью. Эта схема предполагает проведение суммирования в два этапа. На первом этапе все суммируемые значения подразделяются на $n/\log_2 n$ групп, в каждой из которых содержится $\log_2 n$ элементов. Далее для каждой группы вычисляется сумма значений при помощи последовательного алгоритма суммирования. Вычисления выполняются независимо, т.е. могут быть организованы параллельно. Следовательно, для осуществления вычислений на этом этапе необходимо наличие $n/\log_2 n$ процессоров.

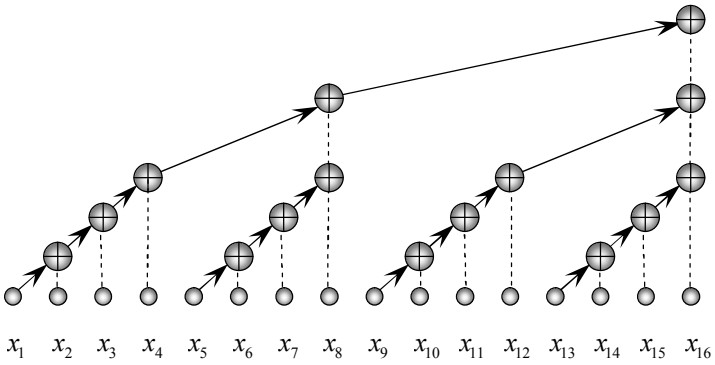


Рис. 1.4. Модифицированная каскадная схема суммирования

На втором этапе при суммировании полученных для отдельных групп $n/\log_2 n$ значений применяется обычная каскадная схема.

Как видно, для рассмотренной схемы получения суммы элементов

последовательности на первом этапе проводится $\log_2 n - 1$ сложений со степенью параллелизма $n / \log_2 n$. На втором этапе выполняется $\log_2 (n / \log_2 n)$ сложений со степенью параллелизма $(n / \log_2 n) \cdot 2^{-l}$, где $l = 1, 2, ..., \log_2 (n / \log_2 n)$. Поэтому

$$T_p \approx \log_2 n - 1 + \log_2 \left(\frac{n}{\log_2 n} \right) \le 2 \log_2 n ,$$

$$S_p \approx \frac{n-1}{2 \log_2 n},$$

а так как $p = \dfrac{n}{\log_2 n}$, то для эффективности этого алгоритма получим $E_p \approx \dfrac{n-1}{2n}$.

1.2. Расчет значений элементов последовательности по рекуррентной формуле

Важным классом рекуррентных вычислений является линейная рекурсия первого порядка, когда элементы последовательности числовых значений $\{x_j\}, j = 1, ..., n$ определяются по соотношениям следующего вида:

$$x_j = a_j x_{j-1} + b_j , \ j = 1, ..., n . \qquad (1.3)$$

Здесь $\{a_j\}, \{b_j\}$ – известные коэффициенты рекуррентной формулы, причем $a_1 = 0$. При последовательном вычислении элементов последовательности требуется $2n$ арифметических операций (рис. 1.5).

В численном анализе, когда необходимо разработать параллельный вычислительный алгоритм, широко применяется метод, известный как ***циклическая редукция***.

Получим формулы метода циклической редукции. Запишем из (1.3)

$$\left.\begin{aligned} x_j &= a_j x_{j-1} + b_j \\ x_{j-1} &= a_{j-1} x_{j-2} + b_{j-1} \end{aligned}\right\},$$

следовательно,

$$x_j = a_j a_{j-1} x_{j-2} + b_j + a_j b_{j-1} = a_j^{(1)}\, x_{j-2} + b_j^{(1)},$$

где $a_j^{(1)} = a_j a_{j-1}$ и $b_j^{(1)} = b_j + a_j b_{j-1}$.

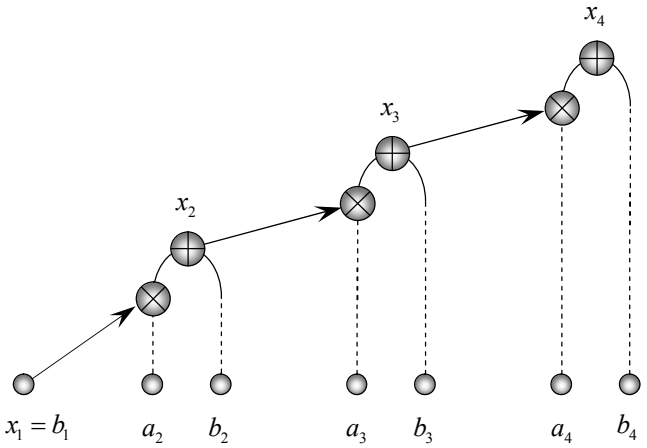


Рис. 1.5. Схема последовательного вычисления элементов по линейной рекурсии первого порядка (1.3)

Последовательно применяя изложенный выше процесс l раз, получим формулы для уровня l:

$$x_j = a_j^{(l)} x_{j-2^l} + b_j^{(l)}\ ,\ j = 1,2,...,n; l = 0,1,...,\log_2 n\,, \qquad (1.4)$$

где

$$a_j^{(l)} = a_j^{(l-1)}\, a_{j-2^{l-1}}^{(l-1)}\,, \qquad (1.5)$$

$$b_j^{(l)} = b_j^{(l-1)} b_{j-2^{l-1}}^{(l-1)} + b_j^{(l-1)} \qquad (1.6)$$

и

$$a_j^{(0)} = a_j, b_j^{(0)} = b_j; j = 1,2,...,n\,.$$

Если индекс любого a_j, b_j, x_j попадает вне диапазона изменения $1 \le j \le n$, то правильный результат применения формул (1.4) – (1.6) получается, если приравнять значение соответствующего элемента нулю. Например, когда $l = \log_2 n$ все значения индексов элементов последовательности $x_{j-2^l} = x_{j-n}$ ($j = 1,2,...,n$), входящих в (1.4), находятся вне обозначенного диапазона $(1,2,...,n)$.

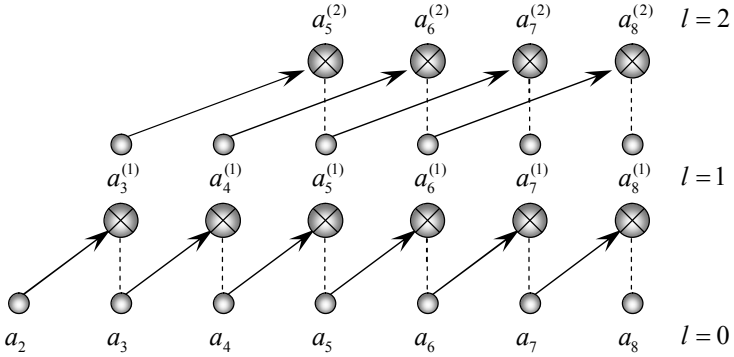


Рис. 1.6. Схема вычисления коэффициентов $a_j^{(l)}$ по формуле (1.5)

В этом случае рекуррентное соотношение (1.4) примет конечный вид:

$$x_j = b_j^{(\log_2 n)}, \quad j = 1,2,...,n. \tag{1.7}$$

Следовательно, чтобы найти значения элементов последовательности методом циклической редукции, необходимо последовательно рассчитывать $a_j^{(l)}$, $b_j^{(l)}$ по (1.5) и (1.6) до тех пор, пока не будет найдено $b_j^{(\log_2 n)}$.

Основная идея циклической редукции заключается в последовательном построении рекуррентных соотношений между каждым вторым, затем каждым четвертым, затем каждым восьмым и т.д. элементами последовательности за счет исключения промежуточных элементов последовательности. Продолжая этот процесс, можно на за-

ключительном этапе (после $\log_2 n$ редукций) получить рекуррентные соотношения, в которых x_j будет связано только со значением элемента последовательности с индексом, лежащим вне известного диапазона, или с нулем.

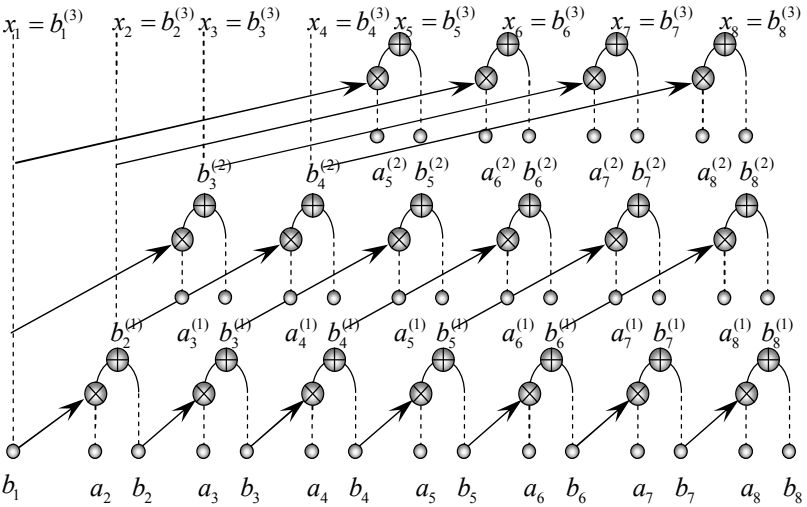


Рис. 1.7. Схема вычисления коэффициентов $b_j^{(l)}$ по формуле (1.6)

На первый взгляд может показаться, что расчетные формулы (1.5), (1.6) не могут применяться параллельно, поскольку они имеют внешне похожее на основное рекуррентное соотношение (1.3) представление. Однако основное различие между формой организации вычислений по формулам (1.3) и (1.5) – (1.6) заключается в том, что в формулах (1.5) – (1.6) все значения $a_{j-2^{l-1}}^{(l-1)}, b_{j-2^{l-1}}^{(l-1)}$ и $a_j^{(l-1)}, b_j^{(l-1)}$ уже были вычислены на предыдущем уровне $(l-1)$, т.е. известны. Поэтому формулы (1.5) – (1.6) имеют явный вид, и вычисления по ним для различных $j = 1, 2, ..., n$ могут выполняться одновременно и независимо (рис. 1.6, 1.7).

Степень параллелизма метода циклической редукции изменяется примерно от n на начальном уровне и до $n/2$ на конечном уровне (рис. 1.6, 1.7).

Приведем теоретические оценки ускорения и эффективности алгоритма для $p = n > 1$:

$$T_1 \approx (t_{mult} + t_{add})n; \quad T_p \approx (2\,t_{add} + t_{mult})\log_2 n;$$

$$S_n \approx \frac{(t_{add} + t_{mult})\,n}{(2\,t_{add} + t_{mult})\log_2 n} \approx \frac{2\,n}{3\log_2 n}\;;\; E_n \approx \frac{2}{3\log_2 n}.$$

2. БАЗОВЫЕ ОПЕРАЦИИ ЛИНЕЙНОЙ АЛГЕБРЫ

В качестве базовых операций линейной алгебры рассмотрим следующие: скалярное произведение векторов, умножение матрицы на вектор и умножение матриц. При проектировании базовых операций линейной алгебры для распределенных вычислений на многопроцессорной технике с локальной памятью большое значение имеет топология вычислительной системы – способ соединения процессорных элементов МВС высокоскоростной компьютерной сетью.

2.1. Вычисление скалярного произведения векторов

Скалярное произведение двух векторов размера n определяется по формуле

$$(\vec{x},\vec{y}) = \sum_{i=1}^{n} x_i y_i \qquad (2.1)$$

и требует n умножений и $n-1$ сложений. Тогда время выполнения последовательного алгоритма на обычном компьютере может быть представлено как

$$T_1 \approx 2t_c n ,$$

где $t_c = \max(t_{add}, t_{mult})$ – время выполнения одной скалярной операции.

Следуя описанной во введении общей методологии создания алгоритмов для параллельных вычислений,

- разобьем задачу на более мелкие подзадачи, а именно мелкозернистая фундаментальная подзадача i запоминает компоненты x_i, y_i и вычисляет произведение $x_i \cdot y_i$;
- коммуникации между подзадачами установим таким образом, чтобы обеспечить суммирование полученных произведений;
- укрупнение подзадач произведем путем объединения n/p (p – предполагаемое число процессоров) мелкозернистых фундаментальных подзадач с сохранением коммуникаций между укрупненными блоками;
- каждая укрупненная подзадача назначается на выполнение одному процессору p-процессорной вычислительной системы.

На рис. 2.1 представлены описанные выше этапы построения параллельной вычислительной процедуры.

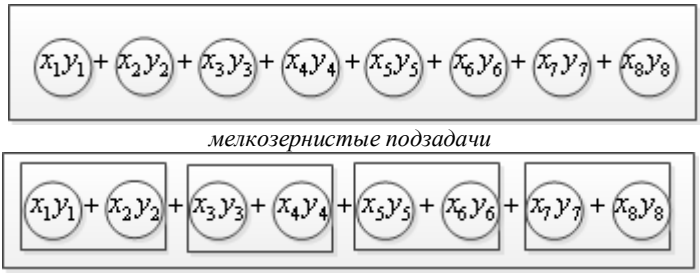


Рис. 2.1. Параллельный алгоритм вычисления скалярного произведения ($n = 8$)

Время, затрачиваемое на получение скалярного произведения на p-процессорной вычислительной системе, без учета затрат на обеспечение коммуникаций определяется по формуле

$$T_p \approx 2t_c \frac{n}{p} + t_{add} \log_2 p .$$

Здесь t_{add} – время арифметической операции суммирования двух чисел с плавающей запятой, первое слагаемое представляет временные затраты на вычисление суммы произведений на каждом процессоре, а второе – суммирование результатов, полученных на каждом процессоре, по алгоритму сдваивания.
Тогда

$$S_p = \frac{T_1}{T_p} \approx \frac{2t_c n}{2t_c \frac{n}{p} + t_{ad} \log_2 p} \approx p \text{ при } n >> p . \qquad (2.2)$$

2.2. Умножение матрицы на вектор

При умножении квадратной матрицы $A \in \mathbb{R}^{n \times n}$ с компонентами $a_{ij}, i, j = 1,...,n$ на вектор $\vec{x} \in \mathbb{R}^n$ компоненты вектора-результата $\vec{y} = A\vec{x}$ вычисляются по формуле

$$y_i = \sum_{j=1}^{n} a_{ij} x_j , \quad 1 \le i \le n , \qquad (2.3)$$

и при последовательном вычислении компонентов вектора $\vec{y}$ временные затраты оцениваются по формуле $T_1 \approx 2t_c n^2$.

Рассмотрим алгоритм умножения матрицы на вектор с точки зрения возможных подходов его распараллеливания. Отметим, что в (2.3):

- операции вычисления каждого y_i независимы и могут быть выполнены параллельно;

- умножение каждой строки на вектор включает независимые операции поэлементного умножения, которые могут быть выполнены параллельно;

- сложение получаемых произведений в каждой операции умножения строки матрицы на вектор может быть выполнено по одной из ранее рассмотренных схем суммирования.

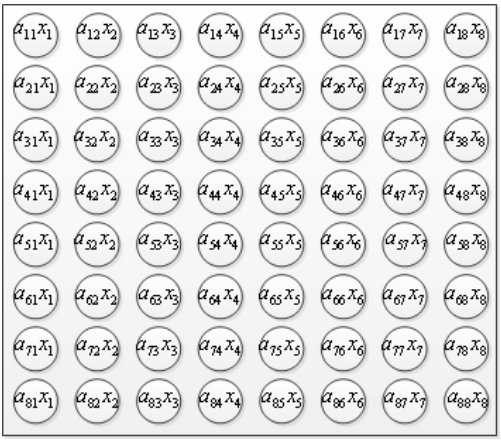


Рис. 2.2. Мелкозернистые подзадачи (n = 8)

Построение алгоритмов для умножения матрицы на вектор проводим по обычной схеме:

1. Задачу разбиваем на мелкие подзадачи, каждая представляет собой вычисление произведения $a_{ij}x_j$ (рис. 2.2). Всего таких подзадач будет n^2. Каждая подзадача (i, j) должна иметь информацию о коэффициенте матрицы a_{ij} и j-й компоненте вектора $\vec{x}$.

2. Для получения значения компоненты вектора $\vec{y}$ с использованием каскадной схемы суммирования необходимо организовать коммуникации между подзадачами $(i,1),(i,2),...,(i,n)$, где $i = 1,...,n$.

3. Укрупнение подзадач проводится на основе одномерной или двумерной стратегии объединения мелкозернистых подзадач. При одномерном укрупнении подзадачи (i, j) объединяются по $\frac{n}{p}$ строк (или столбцов). При двумерном – производится объединение $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ подзадач (рис. 2.3). В этом случае $p = s^2$, где s – натуральное число.

4. Укрупненные подзадачи распределяются по процессорам (см. рис. 2.3).

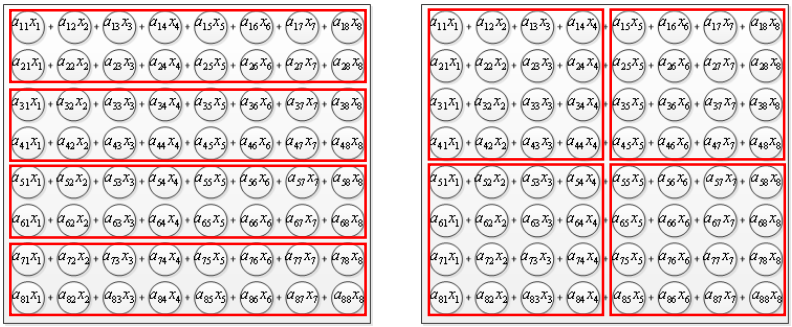


Рис. 2.3. Одномерное (слева) и двумерное (справа) укрупнение мелкозернистых подзадач для $n = 8$

Таким образом, проведенный анализ показывает, что максимально необходимое количество процессоров для осуществления параллельного вычисления произведения матрицы на вектор определяется значением $p = n^2$. Рассмотрим роль количества доступных процессоров при выборе оптимальной топологии межпроцессорных соединений.

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

В этом случае алгоритм можно разбить на подзадачи, в каждой из которых запоминаются значения элемента матрицы a_{ij} и компонен-

ты вектора x_j и рассчитывается их произведение (рис. 2.4). Коммуникации между подзадачами проектируются таким образом, чтобы организовать по каскадной схеме суммирование соответствующих произведений для вычисления отдельной компоненты вектора $\vec{y}$.

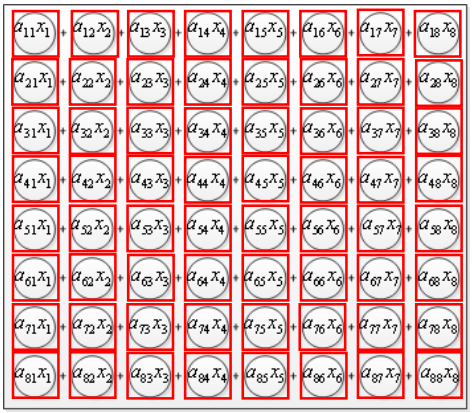


Рис. 2.4. Схема вычислений произведения матрицы на вектор при $p = n^2$ ($n = 8$)

Оценим показатели эффективности алгоритма без учета затрат на обеспечение передачи данных между процессорами. Поскольку

$$T_p \approx t_{mult} + t_{add} \log_2 n \ ,$$

то

$$S_p = \frac{(t_{add} + t_{mult}) n^2}{t_{mult} + t_{add} \log_2 n} \approx \frac{4p}{2 + \log_2 p} \quad (t_{add} \approx t_{mult}) \qquad (2.4)$$

и

$$E_p \approx \frac{4}{2 + \log_2 p} .$$

Для рассматриваемого случая наиболее подходящими будут топологии, в которых обеспечивается быстрая передача данных в каскадной схеме суммирования: полный граф или гиперкуб. Другие топологии приводят к возрастанию коммуникационных затрат из-за удлинения маршрутов передачи данных. Так, например, для линейно

упорядоченных процессоров (топология «линейка» или «кольцо») при выполнении каскадной схемы суммирования длина пути передачи данных определяется величиной $n-1=1+2+4+...+2^{\log_2 n-1}$, в то время как для полного графа – $\log_2 n$.

Рассмотрим другой алгоритм умножения матрицы на вектор на систолическом массиве процессоров ($p=n^2$). В таком массиве процессорные элементы находятся в узлах регулярной решетки, в которой роль ребер играют межпроцессорные соединения. Все процессоры управляются общими командами. В каждом цикле работы любой процессорный элемент (ПЭ) получает данные от соседних ПЭ, выполняет одну команду и передает данные соседям.

$$A^{(1)}=\begin{pmatrix} a_{11} & a_{12} & ... & ... & a_{1n} \\ a_{22} & a_{23} & ... & ... & a_{21} \\ a_{33} & a_{34} & ... & ... & a_{32} \\ ... & ... & ... & ... & ... \\ a_{nn} & a_{n1} & ... & ... & a_{nn-1} \end{pmatrix} \quad X=\begin{pmatrix} x_1 & x_2 & ... & ... & x_n \\ x_2 & x_3 & ... & ... & x_1 \\ x_3 & x_4 & ... & ... & x_2 \\ ... & ... & ... & ... & ... \\ x_n & x_1 & ... & ... & x_{n-1} \end{pmatrix}$$

Размещение исходных данных по процессорным элементам осуществляется следующим образом. Элементы матрицы A распределяют по ПЭ так, что элементы i-й строки матрицы сдвигаются циклически влево на $i-1$ позицию ($n-1$ одиночный сдвиг).

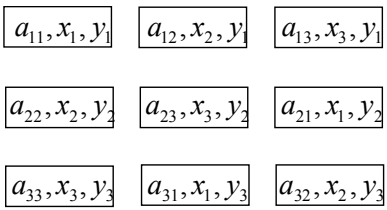


Рис. 2.5. Инициализация алгоритма умножения матрицы на вектор в систолическом массиве процессорных элементов

Вместо вектора $\vec{x}$ рассматривается матрица X, в которой каждый последующий столбец, начиная со второго, получается на основе вектора $\vec{x}$ циклическим сдвигом его компонентов вверх.

Например, исходное распределение данных по процессорным элементам для n=3 представлено на рис. 2.5.

После первого цикла

a_{11}, x_1, y_1	a_{12}, x_2, y_1	a_{13}, x_3, y_1
a_{22}, x_2, y_2	a_{23}, x_3, y_2	a_{21}, x_1, y_2
a_{33}, x_3, y_3	a_{31}, x_1, y_3	a_{32}, x_2, y_3

После второго цикла

a_{12}, x_2, y_1	a_{13}, x_3, y_1	a_{11}, x_1, y_1
a_{23}, x_3, y_2	a_{21}, x_1, y_2	a_{22}, x_2, y_2
a_{31}, x_1, y_3	a_{32}, x_2, y_3	a_{33}, x_3, y_3

После третьего цикла

a_{13}, x_3, y_1	a_{11}, x_1, y_1	a_{12}, x_2, y_1
a_{21}, x_1, y_2	a_{22}, x_2, y_2	a_{23}, x_3, y_2
a_{32}, x_2, y_3	a_{33}, x_3, y_3	a_{31}, x_1, y_3

Каждый процессор производит умножение пары чисел a_{ij} и x_j, записывает результат в y_i (i – номер строки матрицы A или уровня в решетке), передает a_{ij} левому соседу в решетке, а x_j – верхнему. На следующем цикле данные принимаются от соседних процессор-

ных элементов, снова выполняется умножение пары чисел, суммирование результата с y_i и передача данных по ребрам решетки.

После окончания алгоритма на каждом процессорном элементе y_i будет равно сумме $a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3$, однако на промежуточных этапах алгоритма значения y_i в одной строке решетки будет иметь различные значения. Для рассмотренного алгоритма параллельного вычисления произведения матрицы на вектор имеют место оценки:

$$T_p \approx (t_{mult} + t_{add})n = (t_{mult} + t_{add})\sqrt{p}\ ;$$

$$S_p = \frac{T_1}{T_p} \approx \frac{(t_{mult} + t_{add})\, n^2}{(t_{mult} + t_{add})\, n} = n = \sqrt{p}\ ;\quad E_p = \frac{T_1}{pT_p} \approx \frac{1}{\sqrt{p}}, (p = n^2)\ . \quad (2.5)$$

Заметим, что в данном алгоритме не используется каскадная схема суммирования. Сравнивая показатели эффективности рассмотренных алгоритмов (2.4) и (2.5), можно отметить более низкую производительность алгоритма на систолическом массиве процессоров. Например, при p=16 его ускорение уменьшается более чем в 2,5 раза. Количество пересылок данных $2n$ соизмеримо с числом коммуникационных операций первого алгоритма при использовании линейного упорядочивания процессорных элементов (для распределения полученных значений y_i на ПЭ одного уровня потребуется еще $n-1$ передача данных). Наиболее подходящей топологией для вычисления $A\vec{x}$ на систолическом массиве процессоров является топология «тор» или топология «полный граф».

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

Пусть $p = kn$, где $1 < k < n$. В этом случае двумерная декомпозиция задачи может быть осуществлена двумя способами (рис. 2.6): либо с последующим применением обычной каскадной схемы суммирования, либо с использованием модифицированной каскадной схемы.

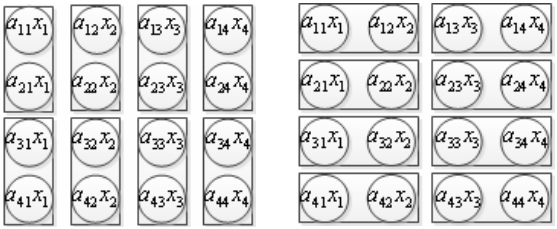


Рис. 2.6. Укрупненные подзадачи ($n = 4$, $k = 2$)

В первом варианте каждая укрупненная подзадача содержит n/k элементов столбца матрицы A и соответствующую компоненту вектора $\vec{x}$. После вычисления произведений производится суммирование по обычной каскадной схеме векторов длиной n/k. Во втором варианте каждой укрупненной подзадаче назначается n/k элементов строки матрицы и n/k компонентов вектора $\vec{x}$ для вычисления произведений. После выполнения операции умножения суммирование произведений осуществляется по модифицированной каскадной схеме. Рассматривая эти варианты, можно заметить, что начальная загрузка каждого процессора увеличивается по сравнению со случаем максимально возможного быстродействия, поскольку необходимо разместить n/k элементов матрицы A и компоненту вектора $\vec{x}$, а во втором варианте еще дополнительно $(n/k)-1$ компоненту вектора $\vec{x}$. Получим оценки для времени выполнения вычислений и ускорения в каждом варианте:

$$T_p^{(1)} \approx t_{mult}\left(\frac{n}{k}\right) + t_{add}\left(\frac{n}{k}\right)\log_2 n = t_{mult}\frac{n^2}{p} + t_{add}\left(\frac{n^2}{p}\right)\log_2 n;$$

$$T_p^{(2)} \approx \left(t_{add} + t_{mult}\right)\left(\frac{n}{k}\right) + t_{add}\log_2 k = \left(t_{add} + t_{mult}\right)\frac{n^2}{p} + t_{add}\log_2\left(\frac{p}{n}\right).$$

$$S_p^{(1)} \approx \frac{\left(t_{add} + t_{mult}\right)n^2}{\frac{n^2}{p}\left(t_{mult} + t_{add}\log_2 n\right)} \approx \frac{2p}{1+\log_2 n}; \qquad (2.6)$$

$$S_p^{(2)} \approx \frac{\left(t_{add} + t_{mult}\right)n^2}{\frac{n^2}{p}\left(\left(t_{add} + t_{mult}\right) + t_{add}\frac{p}{n^2}\log_2\left(\frac{p}{n}\right)\right)} \approx \frac{p}{1+\frac{p}{2n^2}\log_2\left(\frac{p}{n}\right)}. \qquad (2.7)$$

Сравнивая полученные результаты, можно отметить, что второй вариант более предпочтителен, поскольку временные затраты на вычисление вектора $\vec{y}$ существенно меньше. Например,

при $k=\frac{n}{2}$: $T_p^{(1)} \approx 2t_{mult} + 2t_{add}\log_2 n$; $T_p^{(2)} \approx 2t_{mult} + t_{add}(1+\log_2 n)$;

при $k=2$: $T_p^{(1)} \approx \frac{n}{2}t_{mult} + \frac{n}{2}t_{add}\log_2 n$; $T_p^{(2)} \approx \frac{n}{2}t_{mult} + t_{add}\left(1+\frac{n}{2}\right)$.

В случае, когда $p=2n$ (k=2), можно организовать конвейерное умножение матрицы на вектор на многопроцессорной вычислительной системе с распределенной памятью (см. рис. 2.7). Возможность конвейерной обработки данных связана с разделением процесса выполнения вычислений (умножения матрицы на вектор) на последовательные этапы. В рассматриваемом случае это умножение коэффициентов матрицы на компоненты вектора, вычисление суммы произведений по алгоритму каскадной схемы суммирования (алгоритму сдваивания, см. главу 1). При конвейерной обработке данных выполнение следующего этапа начинается только после окончания предыдущего, однако выполнение всех этапов над отдельными компонентами данных, участвующих в общем процессе обработки, можно совмещать.

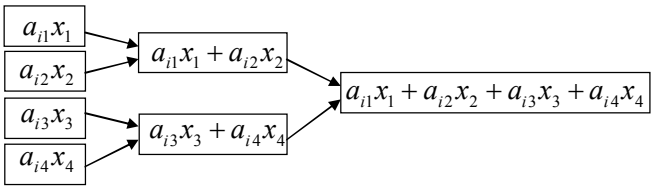


Рис. 2.7. Конвейерная схема вычисления произведения матрицы на вектор (n = 4, p = *2n–1*)

Для обеспечения такого вычислительного процесса множество процессоров разбивается на непересекающиеся процессорные группы $Q=\{Q_0, Q_1, ..., Q_m\}$, где $m=\log_2 n$. Группа процессоров Q_0, объединяющая n процессорных элементов, используется для реализации

поэлементного умножения i-й строки матрицы A на вектор $\vec{x}$. На каждом такте работы конвейера следующая строка матрицы A поступает на процессоры группы Q_0. Каждая группа процессоров Q_l $(1 \le l \le m)$ состоит из $\frac{n}{2^l}$ процессорных элементов и предназначена для выполнения l-й итерации алгоритма сдваивания. Следует отметить, что в один момент времени конвейер обрабатывает $\log_2 n + 1$ строк матрицы A.

Получим оценку показателей эффективности и ускорения построенного вычислительного процесса:

$$T_p \approx \left(t_{mult} + t_{add} \log_2 n\right) + \max\left(t_{mult}, t_{add}\right)(n-1).$$

Здесь первое слагаемое определяет затраты на загрузку конвейера (время, необходимое для задействования всех этапов конвейерного способа обработки данных), второе – временной интервал одного такта работы конвейера (время работы одного этапа конвейера). Тогда

$$S_p \approx \frac{2p}{1 + \frac{\log_2 n}{n}}; \quad E_p \approx \frac{1}{1 + \frac{\log_2 n}{n}}. \tag{2.8}$$

Сравнивая полученные показатели с (2.7), получаем, что при конвейерной реализации умножения матрицы на вектор временные затраты выше. Однако рассмотренный способ организации параллельных вычислений имеет преимущества: меньшее количество пересылок данных между процессорами и возможность получения части результатов до окончания всего вычислительного процесса.

Характер организации межпроцессорных коммуникаций, представленных на рис. 2.7, указывает, что наиболее подходящей топологией многопроцессорной вычислительной системы будет полное двоичное дерево высотой $\log_2 n + 1$. Количество передач данных при такой топологии многопроцессорной вычислительной системы – $n + \log_2 n$.

Организация параллельных вычислений при $p = n$

В случае использования n процессоров может быть реализован один из следующих подходов при вычислении произведения $A\vec{x}$:

- с помощью линейных комбинаций;
- на основе алгоритма скалярных произведений.

По первому алгоритму вектор $\vec{y} = A\vec{x}$ определяется из

$$\vec{y} = \sum_{i=1}^{n} x_i \cdot \vec{a}_i \,, \tag{2.9}$$

где $\vec{a}_i$ – i-й столбец матрицы A. При такой схеме вычислений очевидно, что произведения – векторы $x_i\vec{a}_i$ – рассчитываются независимо с максимальным параллелизмом. На каждый i-й процессорный элемент МВС необходимо распределить x_i и $\vec{a}_i$. После вычисления произведения числа на вектор выполняются сложения по методу сдваивания, применяемого здесь к векторам.

По второму алгоритму компонента вектора $\vec{y}$ вычисляется как скалярное произведение векторов

$$y_i = (\vec{\alpha}_i, \vec{x}), \tag{2.10}$$

где $\vec{\alpha}_i$ – i-я строка матрицы A. На i-й процессорный элемент необходимо распределить два вектора – $\vec{\alpha}_i$ и $\vec{x}$. Затем каждый процессор вычисляет скалярное произведение этих векторов с максимальным параллелизмом, причем не требуется применения алгоритма сдваивания.

Заметим, что:

- различие представлений (2.9) и (2.10) можно интерпретировать как различие двух способов доступа к данным. В алгоритме линейных комбинаций выборка элементов матрицы A проводится по столбцам, а при использовании скалярных произведений – по строкам;

- применение алгоритмов (2.9) и (2.10) для n-процессорной системы дает различные временные затраты на инициализацию (подготовку) вычислительного процесса: по первому алгоритму на каждый процессорный элемент требуется переслать $n+1$ число, по второму – $2n$;

- после выполнения (2.9) результат – вектор $\vec{y}$ – размещается на одном процессоре, а по алгоритму (2.10) i-я компонента вектора $\vec{y}$ получена на процессоре с соответствующим номером. Достоинства и недостатки такого распределения результата умножения матрицы на вектор зависят от его использования при последующих вычислениях.

Рассмотрим другой возможный способ организации параллельных вычислений, который заключается в построении конвейерной схемы для операции умножения строки матрицы на вектор при применении скалярного произведения путём расположения всех имеющихся процессоров в виде линейной последовательности (топология «линейка»).

Представим множество процессоров в виде линейной последовательности $Q = \{ПЭ_1, ПЭ_2, ..., ПЭ_n\}$, где процессорный элемент $ПЭ_j$ $(1 \le j \le n)$ используется для умножения элементов j-го столбца матрицы и j-го элемента вектора $\vec{x}$ (рис. 2.8).

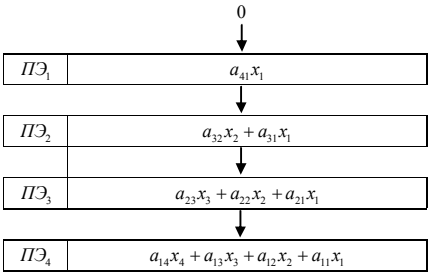


Рис.2.8. Распределение данных на линейной последовательности процессорных элементов при реализации конвейерной схемы вычислений для $n = 4$

Выполнение вычислений на каждом процессоре состоит в следующем:

- на каждом процессоре запрашивается очередной (i-й) элемент j-го столбца матрицы A;

- выполняется умножение элементов a_{ij} и x_j;

- запрашивается результат вычислений предшествующего процессора и копируется в переменную S;

- выполняется сложение значений $S \leftarrow S + a_{i\,j}\, x_j$;

- получаемый результат S пересылается следующему процессору.

В результате работы построенной конвейерной системы на последнем процессорном элементе линейной последовательности процессоров будут последовательно ($i = 1,2,...,n$) получаться компоненты вектора $\vec{y}$. При инициализации вычислений на каждом процессоре $ПЭ_j$ нужно разместить x_j.

Для однородности вычислений на первый процессорный элемент $ПЭ_1$, у которого нет предшествующего процессора, на каждом такте работы конвейера целесообразно пересылать $S = 0$.

Выполним оценку показателей эффективности алгоритма. Загрузка конвейера, в процессе которой будет произведено умножение первой строки матрицы A на вектор $\vec{x}$, завершится после $n+1$ арифметической операции. Результаты скалярных произведений следующих строк A на $\vec{x}$ будут получаться на каждом такте работы конвейера (на каждом такте проводится одно сложение и одно умножение двух действительных чисел).

В итоге время, необходимое на вычисление произведения матрицы на вектор без учета затрат на передачу данных, определится следующим образом:

$$T_p \approx \left(t_{mult} + t_{add}\, p\right) + \left[\left(t_{mult} + t_{add}\right)\left(p-1\right)\right],\ p = n.$$

Заметим, что применение конвейерной схемы организации вычислений на n-процессорном мультикомпьютере с линейной топологией соединения узлов дает большее время, чем по обычной схеме скалярных произведений – $\left(t_{mult} + t_{add}\right) p$. Полезность использования конвейерного способа организации матрично-векторных вычислений состоит в уменьшении количества передаваемых данных и в более раннем появлении части результатов вычислений. Ускорение и эффективность конвейерной схемы рассчитываются следующим образом:

$$S_p = \frac{T_1}{T_p} \approx \frac{\left(t_{mult} + t_{add}\right) p^2}{p t_{mult} + (2p-1) t_{add}} \approx \frac{2p^2}{3p\text{-}1}, \qquad (2.11)$$

$$E_p = \frac{T_1}{pT_p} \approx \frac{(t_{mult} + t_{add})p}{pt_{mult} + (2p-1)t_{add}} \approx \frac{2p}{3p\text{-}1}.$$

Необходимая топология вычислительной системы для выполнения конвейерной схемы – это линейно упорядоченное множество процессорных элементов (линейка процессоров).

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

При ограниченном наборе процессорных элементов организация вычислительной схемы параллельного умножения матрицы на вектор может быть представлена следующим образом:

- на каждый из имеющихся процессорных элементов пересылается вектор $\vec{x}$ и $k = n / p$ строк матрицы A;

- операция умножения строк матрицы на вектор выполняется при помощи обычного последовательного алгоритма.

Этот алгоритм эквивалентен умножению матрицы на вектор в блочной форме

$$A\vec{x} = \begin{bmatrix} A_1 \\ \dots \\ A_p \end{bmatrix} \vec{x} = \begin{bmatrix} A_1\vec{x} \\ \dots \\ A_p\vec{x} \end{bmatrix},$$

где матрица A_i $(i = 1, 2, ..., p)$ составлена из n / p строк матрицы A. При принятом распределении данных по процессорам вычисления произведений $A_i\vec{x}$ осуществляются с максимальным параллелизмом, при этом не имеет значения, каким образом выполняются умножения $A_i\vec{x}$ на каждом процессоре – по алгоритму скалярных произведений или по алгоритму линейных комбинаций. Аналогичные соображения применимы к разбиению A на группы столбцов.

Следует, однако, отметить, что размер матрицы может не оказаться кратным количеству процессоров, и тогда строки матрицы не могут быть разделены поровну между вычислительными узлами. В этих ситуациях нужно отступить от требования равномерности загрузки процессоров и для получения более простой вычислительной схемы принять правило, что размещение данных на процессорах осуществляется только построчно (т.е. элементы одной строки не могут быть

распределены между различными вычислительными узлами). Однако неравномерность загрузки процессоров снижает эффективность использования МВС, поскольку процессоры с меньшей загрузкой после завершения своей вычислительной работы будут простаивать, а общая длительность решения задачи определяется временем работы наиболее загруженного процессора. Поэтому необходимо стремиться распределить строки (или столбцы) между процессорами по возможности равномерно. Проблема балансировки загрузки процессоров относится к числу одной из важнейших задач параллельного программирования.

Подсчитаем вычислительные временные затраты для рассматриваемого случая

$$T_p \approx \left(t_{mu} + t_{ad}\right)\left\lceil \frac{n}{p} \right\rceil p ,$$

где $\lceil\ \rceil$ – округление до ближайшего большего целого.

Тогда

$$S_p \approx \frac{\left(t_{mult} + t_{add}\right) n^2}{\left(t_{mult} + t_{add}\right)\left\lceil \frac{n}{p} \right\rceil n} = \frac{n}{\left\lceil \frac{n}{p} \right\rceil}, \qquad (2.12)$$

$$E_p \approx \frac{n}{p\left\lceil \frac{n}{p} \right\rceil}.$$

С учетом характера выполняемых межпроцессорных взаимодействий в предложенной вычислительной схеме в качестве возможной топологии может быть использована коммуникационная структура процессоров в виде звезды (коммуникационная структура «хозяин/работник», см. рис. 0.3 Введения). Управляющий процессор обеспечивает загрузку вычислительных узлов исходными данными и собирает результаты проведенных вычислений.

Рассмотрим другой способ распределения данных между процессорными элементами, который использует двумерное укрупнение мелкозернистых подзадач (рис. 2.3). Пусть $p = s^2$, $s \in \mathbb{N}$ и $n \bmod s = n \bmod \sqrt{p} = 0$. В этом случае на каждом процессорном

элементе помещается $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ коэффициентов матрицы A и $\frac{n}{\sqrt{p}}$ компонентов вектора $\vec{x}$ (рис. 2.9). Каждый процессор локально выполняет $\frac{n^2}{p}$ умножений и $\frac{n^2}{p}$ сложений полученных произведений. Затем по алгоритму сдваивания осуществляется суммирование $\frac{n}{\sqrt{p}}$ результатов с процессорных элементов, расположенных в одном ряду, для определения $\frac{n}{\sqrt{p}}$ компонентов вектора $\vec{y}$.

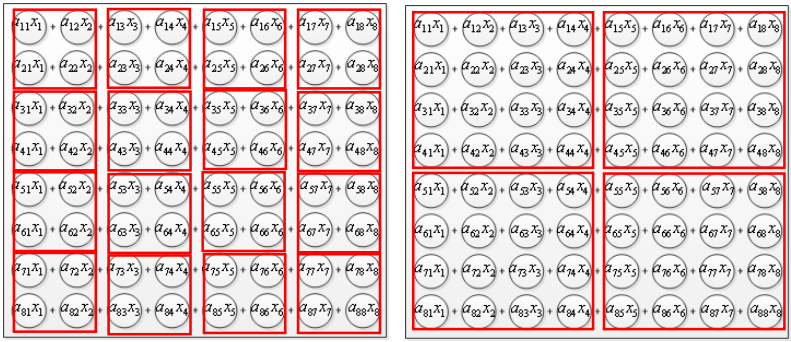


Рис. 2.9. Двумерное распределение данных между процессорными элементами. $n = 8$, $p = 16$ (слева), $p = 4$ (справа)

Время, затраченное на выполнение вычислений, определяется как

$$T_p \approx \frac{n^2}{p} t_{mult} + \frac{n^2}{p} t_{add} + t_{add} \frac{n}{\sqrt{p}} \log_2 \left(\frac{n}{\sqrt{p}} \right).$$

Тогда

$$S_p \approx \frac{p}{1 + \frac{\sqrt{p}}{2n} \log_2 \left(\frac{n}{\sqrt{p}} \right)}; \qquad (2.13)$$

$$E_p \approx \frac{1}{1+\frac{\sqrt{p}}{2n}\log_2\left(\frac{n}{\sqrt{p}}\right)}.$$

Сравнивая (2.12) и (2.13), можно заметить, что ускорение алгоритма при двумерном укрупнении мелкозернистых подзадач несколько ниже, чем при одномерном, что обусловлено использованием алгоритма сдваивания. Кроме того, из-за необходимости применения каскадной схемы суммирования при двумерной декомпозиции потребуются коммуникационные операции между процессорными элементами, обрабатывающими компоненты одной строки матрицы A. Заметим, что затраты на инициализацию обоих алгоритмов примерно одинаковы.

2.3. Умножение квадратных матриц

Матричное умножение требует выполнения большого количества арифметических операций.

Пусть A, B, C – квадратные матрицы $n \times n$, $C = AB$. Тогда компоненты матрицы C рассчитываются по следующей формуле:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}\ ,\ i, j = 1,...,n\ . \qquad (2.14)$$

Из (2.14) следует, что для вычисления одного элемента матрицы C необходимо n умножений и n сложений. Учитывая общее количество таких элементов, получим, что операция умножения матриц потребует выполнения n^3 скалярных умножений и n^3 сложений на обычном последовательном компьютере:

$$T_1 \approx \left(t_{mult} + t_{add}\right) n^3\ .$$

Произведение матриц может рассматриваться как n^2 независимых скалярных произведений либо как n независимых произведений матрицы на вектор. В каждом случае используются различные алгоритмы.

Эффективность умножения матриц может быть улучшена путем изменения порядка циклов ijk в (2.14). Каждый язык программирования характеризуется различным способом хранения в памяти компьютера элементов массивов. На языке программирования высокого уровня Fortran элементы матриц располагаются последовательно в памяти по столбцам матрицы, т.е. $\{a_{11}, a_{21}, ..., a_{n1}, a_{12}, a_{22}, ..., a_{n2}, a_{13}, ..., a_{nn}\}$.

Размещаемые в кэш-память элементы матриц A, B, C эффективно используются, когда доступ к ним в алгоритме умножения матриц осуществляется последовательно с переходом к соседней ячейке памяти (рис. 2.10).

```
!цикл i-j-k
 do i = 1, n
  do j = 1, n
   do k = 1, n
    c(i, j) = c(i, j)+a(i, k)*b(k, j)
   end do
  end do
end do
```

```
!цикл k-j-i
    do k = 1, n
      do j = 1, n
        do i = 1, n
          c(i, j) = c(i, j)+a(i, k)*b(k, j)
        end do
      end do
    end do
```

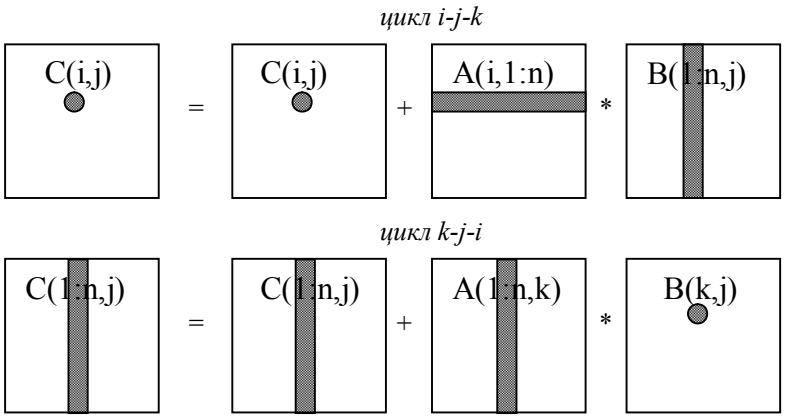


Рис. 2.10. Схема доступа к элементам матриц

Другим подходом для повышения быстродействия умножения матриц является использование блочного представления матриц.

$$C = \begin{bmatrix} C_{11} & \dots & C_{1N} \\ \dots & \dots & \dots \\ C_{N1} & \dots & C_{NN} \end{bmatrix} = AB = \begin{bmatrix} A_{11} & \dots & A_{1N} \\ \dots & \dots & \dots \\ A_{N1} & \dots & A_{NN} \end{bmatrix} \begin{bmatrix} B_{11} & \dots & B_{1N} \\ \dots & \dots & \dots \\ B_{N1} & \dots & B_{NN} \end{bmatrix},$$

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} , \qquad (2.15)$$

где A_{ik}, B_{kj}, C_{ij} – матрицы размером $(n/N) \times (n/N)$.

Чтобы понять, как достигается ускорение вычислений при таком представлении матриц, необходимо знать, на что тратится машинное время при выполнении программы и как функционирует память компьютера.

Память компьютера обычно устроена иерархически и состоит из различных видов памяти с очень быстрой, дорогой и поэтому малого объема памятью на вершине иерархии и медленной, дешевой и большого объема памятью в ее основании:

Рис. 2.11. Иерархическая структура памяти компьютера

Арифметические и логические операции выполняются только с данными, размещенными в регистрах. С одного уровня памяти данные могут перемещаться на соседние уровни, причем скорость передачи данных снижается к основанию, и время обмена данными на нижних уровнях иерархической памяти на порядки может превосходить время, за которое выполняются арифметические и логические операции на регистрах. Поэтому эффективность и производительность вычислений в значительной степени зависят от того, как будет организована передача данных из основания иерархической памяти в ее регистры. Особенно это становится актуальным при больших объемах обрабатываемых данных, например, когда перемножаемые мат-

рицы могут разместиться целиком лишь в большом и медленном уровне иерархической памяти.

Рассмотрим, как может повлиять на разработку алгоритма наличие кэш-памяти (высокоскоростной оперативной памяти компьютера). Обмен данными между основной памятью и кэш-памятью требует больше времени, чем между кэш-памятью и регистрами. Поэтому, если какие-то данные попали в быструю память, нужно стремиться их использовать максимальным образом.

Предположим, что размеры оперативной памяти достаточно велики, чтобы вместить три массива матриц A, B, C, а кэш может вмещать M чисел с плавающей точкой, причем $2n < M << n^2$. Последнее неравенство означает, что в кэш-памяти могут быть размещены два столбца или две строки перемножаемых матриц, но матрица целиком размещена быть не может. Предположим также, что программист имеет возможность управлять движением данных.

Рассмотрим обычную версию матричного умножения (2.14, цикл ijk) с целью показать движение данных между уровнями памяти и оценить количество передач данных:

```
do i = 1, n
 ! считываем строку i матрицы A в быструю память
 do j = 1, n
  ! считываем элемент cij в быструю память
  ! считываем столбец j матрицы B в быструю память
  do k = 1, n
   c(i, j) = c(i, j)+a(i, k)*b(k, j)
  end do
  ! записываем cij из быстрой памяти в медленную
 end do
end do
```

Выполним подсчет числа обмена данными между основной и кэш-памятью Γ_1. Для передачи элементов матрицы A потребуется n^2 обращений, B – n^3, C – $2n^2$. В итоге получим $\Gamma_1 = n^3 + 3n^2$ обращений.

Теперь покажем, что подход, основанный на блочных скалярных произведениях (2.15), приводит к более эффективному использова-

нию кэш-памяти. Рассмотрим блочную версию матричного умножения:

```
do i = 1, N
  do j = 1, N
    ! считываем блок Cij в быструю память
    do k = 1, N
      ! считываем блок Aik в быструю память
      ! считываем блок Bkj в быструю память
      c(i, j) = c(i, j)+a(i, k)*b(k, j)
    end do
    ! записываем Cij из быстрой памяти в медленную
  end do
end do
```

По этому алгоритму число передаваемых данных будет следующим: для матрицы A – $N^3(n^2/N^2)=Nn^2$, для матрицы B – также Nn^2, а для чтения или записи блоков матрицы C – $N^2(n^2/N^2)=n^2$. Всего получается число передаваемых по маршруту «оперативная память – кэш-память» данных $\Gamma_2=2(N+1)n^2\approx 2Nn^2$. Поэтому, чтобы минимизировать число обращений, нужно взять как можно меньшее значение N. Но N подчиняется ограничению $M\geq(3n^2/N^2)$, которое означает, что в кэш-памяти, размер которой M чисел, должно разместиться по одному блоку матриц A,B,C. Отсюда получаем $N\approx n\sqrt{3/M}$ и $\Gamma_2\approx 2n^3\sqrt{3/M}$. Тогда отношение $\Gamma_1/\Gamma_2\approx\sqrt{M}/(2\sqrt{3})>1$, и можно сделать вывод, что блочные скалярные произведения имеют существенное преимущество.

Процедура параллельного умножения матриц легко может быть построена на основе полученных выше алгоритмов скалярного умножения векторов или матрично-векторного умножения для МВС. Введение макроопераций типа **dot** (скалярное произведение векторов) или **gaxpy** (умножение матрицы на вектор плюс вектор) значительно упрощает проблему выбора эффективного способа распараллеливания вычислений, позволяет использовать типовые параллельные методы выполнения макроопераций в качестве конструктивных

элементов при разработке параллельных способов решения сложных вычислительных задач.

Воспользуемся общей процедурой построения параллельных методов решения задачи матричного умножения:

1. *Декомпозиция*. На основе проведенного выше анализа выберем в качестве отдельного фрагмента вычислительной задачи (фундаментальной подзадачи) вычисление произведения $a_{ik}b_{kj}$. Количество таких подзадач равно n^3, и их можно представить в виде трехмерного массива (рис. 2.12). Данные, необходимые для вычисления произведения, определяются компонентами матриц a_{ik} и b_{kj}.

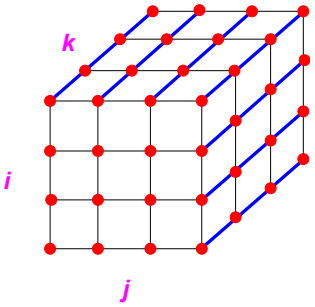


Рис. 2.12. Фундаментальные подзадачи и коммуникации между ними

2. *Проектирование коммуникаций*. Для вычисления значения элемента матрицы c_{ij} требуется обеспечение коммуникаций между подзадачами, в которых производится расчет произведений $a_{ik}b_{kj}$, ($k = 1,...,n$), для последующего суммирования результатов (рис. 2.12).

3. *Укрупнение*. Для имеющихся n^3 фундаментальных подзадач могут быть выбраны следующие естественные стратегии их объединения в p ($p << n^3$) блоков (рис. 2.13):

- одномерное укрупнение по строкам: в блок объединяется $(n/p)\times n\times n$ подзадач. Для выполнения i-й укрупненной подзадачи требуются матрица A_i размерности $(n/p)\times n$ и все элементы матрицы B.

- одномерное укрупнение по столбцам: укрупненный блок содержит $n \times (n / p) \times n$ фундаментальных подзадач. В этом случае для вычислений необходимы целиком матрица A и матрица B_j, состоящая из $n \times \left(n / p\right)$ элементов.

- двумерное укрупнение: производится объединение $\left(n / \sqrt{p}\right) \times (n / \sqrt{p}) \times n$ фундаментальных подзадач. Здесь реализуется алгоритм, основанный на блочном представлении матриц: $C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik} B_{kj}$, $i, j = 1,..., \sqrt{p}$. Блок (i, j), полученный после укрупнения, должен содержать блоки A_{ik}, B_{kj}, $k = 1,..., \sqrt{p}$.

- трехмерное укрупнение: каждая укрупненная подзадача объединяет $(n / \sqrt[3]{p}) \times (n / \sqrt[3]{p}) \times (n / \sqrt[3]{p})$ мелкозернистых подзадач.

Рассматривая представленные выше стратегии укрупнения, нужно отметить одну важную особенность: при использовании одномерной декомпозиции максимально можно задействовать лишь n процессорных элементов, при двумерной – n^2, а при трехмерной – n^3. Тогда если алгоритм хорошо масштабируется, то фиксированный объем информации за меньшее время можно обработать с использованием трехмерной декомпозиции, поскольку для вычислений привлекается большее количество процессорных элементов.

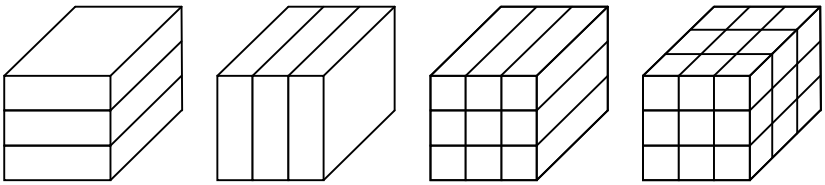


Рис. 2.13. Стратегии укрупнения мелкозернистых подзадач:
1D – одномерное укрупнение, 2D – двумерное укрупнение,
3D – трехмерное укрупнение

4. *Планирование вычислений*. На этом этапе разработки параллельного алгоритма необходимо определить, где, на каких процессорных элементах будут вычисляться укрупненные подзадачи.

Проведя расчет временных затрат без учета обменных операций между процессорами (заметим, что при вычислении произведения матриц передача данных между процессорами необходима только для трехмерного укрупнения) для каждой из построенных процедур параллельного умножения матриц, получим $T_p = (t_{mult} + t_{add})n^3 / p$. Тогда

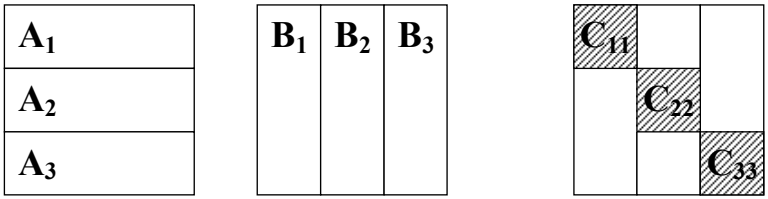
$$S_p = \frac{T_1}{T_p} \approx \frac{(t_{mult} + t_{add})n^3}{(t_{mult} + t_{add})n^3 / p} = p . \qquad (2.16)$$

Алгоритмы, рассмотренные выше, требуют значительных ресурсов памяти многопроцессорной вычислительной системы для хранения блоков матриц A, B, C. Рассмотрим некоторые алгоритмы, позволяющие сократить затраты памяти при проведении параллельного умножения матриц. Эти алгоритмы основывыются на разбиении матриц на блоки в соответствии с числом используемых процессоров и организации обменов этими блоками между процессорными элементами.

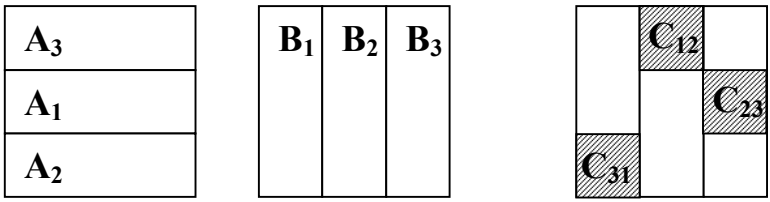
Простой блочный алгоритм

Распределение данных между процессорами производится следующим образом. Матрица A распределяется по процессорам блоками, содержащими $(n / p) \times n$ элементов, B – $n \times (n / p)$ элементов. Далее проводится матричное умножение соответствующих блоков, и определяются блоки матрицы C размером $(n / p) \times (n / p)$, стоящие на главной диагонали.

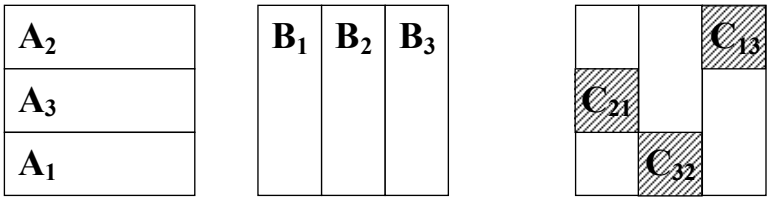
После этого на каждой следующей итерации производится обмен блоками матрицы A между процессорными элементами циклическим сдвигом по уменьшению номера процессора (или увеличению) и вычисления произведений блочных матриц. После p итераций расчет матрицы C заканчивается. Такой способ параллельного умножения матриц может быть использован при одномерном укрупнении подзадач.



шаг 1



шаг 2



шаг 3

Заметим, что для выполнения этого алгоритма оперативная память каждого ПЭ должна вмещать не менее $3n^2 / p$ чисел с плавающей точкой. Тогда общее количество пересылаемых элементов матрицы A:

$$p\,(\text{этапов}) \times \frac{n^2}{p} (\text{размер блока } A_i) = n^2 .$$

Предполагается, что пересылка блоков $A_i, i = 1,..., p$ между процессорными элементами на каждом шаге осуществляется одновременно.

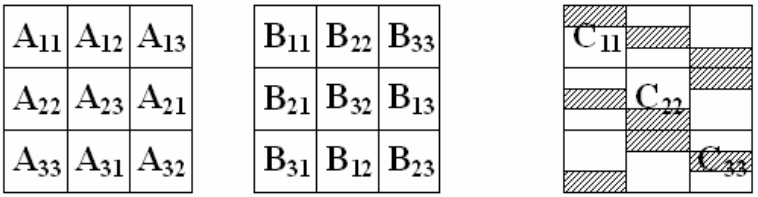
Алгоритм Кэннона

Для этого алгоритма количество процессоров $p = s^2$, где s – натуральное число. Матрицы A и B делятся на p блоков согласо-

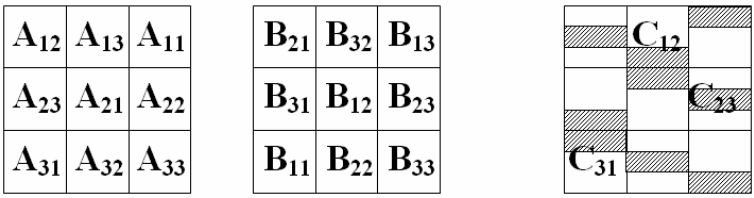
ванных размеров: $A_{ij}, B_{ij}, i, j = 1,..., \sqrt{p}$. На каждый процессорный элемент p_{ij} рассылаются соответствующие матрицы A_{ij}, B_{ij}. Находится их произведение. Затем на каждой итерации производится: пересылка матрицы A_{ij} левому соседу, а матрицы B_{ij} – верхнему в двумерной решетке процессоров, в которой первый и последний узлы как в ряду решетки, так и в ее колонке имеют прямое соединение; вычисление произведения полученных матриц. После выполнения $\sqrt{p}$ итераций получается матрица C_{ij} на каждом процессорном элементе.

В алгоритме Кэннона в оперативной памяти каждого процессорного элемента также должны размещаться $3n^2 / p$ чисел с плавающей точкой, однако объем пересылаемой между процессорными элементами информации намного меньше. Действительно, количество чисел, пересылаемых за все время выполнения алгоритма (здесь также предполагается, что на каждом шаге алгоритма данные передаются одновременно), равно

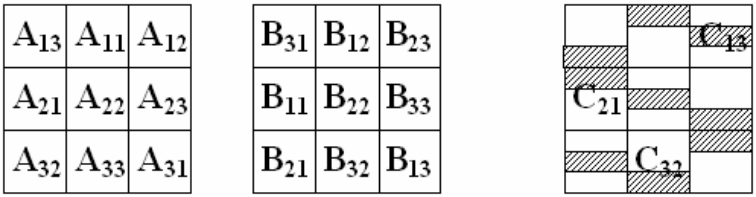
s (этапов)*2(блока)* n^2 / s^2 (размер одного блока)= $2n^2 / \sqrt{p}$.



шаг 1



шаг 2



шаг 3

Алгоритм Фокса

Пусть $p = s^2$. Представим исходные матрицы A, B и результат C в виде наборов квадратных блоков размером $n/s \times n/s$.

$$\begin{pmatrix} A_{11} & A_{12} & ... & A_{1s} \\ A_{21} & A_{22} & ... & A_{2s} \\ ... & ... & ... & ... \\ A_{s1} & A_{s2} & ... & A_{ss} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} & ... & B_{1s} \\ B_{21} & B_{22} & ... & B_{2s} \\ ... & ... & ... & ... \\ B_{s1} & B_{s2} & ... & B_{ss} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & ... & C_{1s} \\ C_{21} & C_{22} & ... & C_{2s} \\ ... & ... & ... & ... \\ C_{s1} & C_{s2} & ... & C_{ss} \end{pmatrix},$$

$$C_{ij} = \sum_{m=1}^{s} A_{im} B_{mj}, \; i, j = 1,..,s.$$

Для организации параллельных вычислений предположим, что процессоры образуют логическую прямоугольную решетку $s \times s \, (p_{ij}, i, j = 1,...,s)$.

В соответствии с алгоритмом Фокса каждый процессорный элемент p_{ij} отвечает за вычисление C_{ij}. В ходе вычислительного процесса на каждом p_{ij} размещаются вычисляемое C_{ij}, исходная матрица A_{ij}, A'_{ij} и B'_{ij} – блоки матриц A, B, получаемые в ходе выполнения алгоритма.

На этапе инициализации производится рассылка на каждый вычислительный узел p_{ij} блоков A_{ij}, B_{ij} и присваивание нулевых значений элементам матрицы C_{ij}. При проведении вычислений на каждой итерации l $(1 \le l \le s)$ выполняются следующие действия:

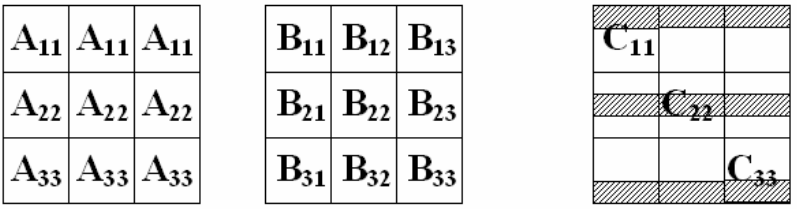
- для каждого ряда процессорной решетки i $(1 \le i \le s)$ определяется значение индекса m по формуле $m = (i + l - 1) \bmod s + 1$;

- блок A_{im} пересылается на все процессорные элементы ряда i и производится присваивание $A'_{ij} = A_{im}, j = 1, ..., s$;

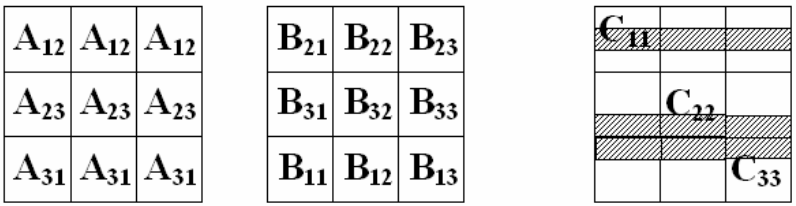
- полученные в результате пересылок матрицы A'_{ij}, B'_{ij} каждого процессорного элемента p_{ij} перемножаются и прибавляются к матрице C_{ij}:

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij};$$

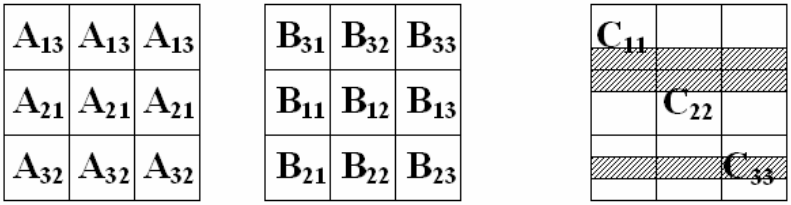
- матрицы B'_{ij} каждого процессорного элемента p_{ij} пересылаются $p_{i-1\,j}$, являющимися соседями сверху в колонках процессорной решетки (ПЭ $p_{1\,j}$ отправляет данные ПЭ p_{sj}).



шаг 1



шаг 2



шаг 3

Приведенный параллельный метод матричного умножения в случае, если n нацело делится на s, приводит к равномерной загрузке процессоров:

$$T_p \approx \left(t_{mult} + t_{add}\right)\left(\frac{n}{s}\right)^3 \cdot s = \frac{\left(t_{mult} + t_{add}\right) n^3}{p};$$

$$S_p \approx \frac{\left(t_{mult} + t_{add}\right) n^3}{\left(t_{mult} + t_{add}\right) n^3 / p} = p. \qquad (2.17)$$

Однако блочное представление матриц в рассмотренных выше алгоритмах умножения матриц приводит к некоторому увеличению объёма пересылаемых между процессорными элементами данных. Так, в алгоритме Фокса на каждой итерации этапа вычислений на p_{ij} процессорный элемент дополнительно передаётся два массива данных общим объёмом $2(n/s)^2$. Тогда на каждый вычислительный узел необходимо отправить (без учета затрат на размещение блоков при подготовке к выполнению алгоритма)

$$V_{ij} = 2(n/s)^2 s = 2n^2 / \sqrt{p} = \frac{2n^2}{\sqrt{p}}$$

данных. Объем пересылаемых данных может быть уменьшен, если использовать строковое для A и столбцовое для B блочное распределение данных по процессорным элементам.

Блочное представление матриц при параллельном вычислении матричных произведений имеет ряд преимуществ:

- пересылка данных является распределённой во времени, что позволяет совмещать вычисления и передачу данных;

- блочная структура может быть использована при решении систем линейных уравнений с разреженной матрицей.

Рассмотренный метод распределения данных по вычислительным узлам многопроцессорной системы является примером геометрического принципа распараллеливания, или декомпозиции по данным. Такая идея широко используется при разработке параллельных методов решения сложных задач, поскольку во многих случаях может приводить к значительному снижению потоков пересылаемых данных за счёт локализации на процессорах существенно информационно зависимых частей алгоритмов, например, при численном решении краевых задач.

2.4. Библиотека базовых подпрограмм линейной алгебры

Для реализации операций линейной алгебры на таких вычислительных системах, как CrayT3E, IBM SP2, Intel Paragon, SGI Power Challenge, TM CM-5, Linux-кластерах или любой другой системе, на которой установлена библиотека MPI (Message Passing Interface) или PVM (Parallel Virtual Machine), разработана специализированная библиотека PBLAS (основные параллельные подпрограммы линейной алгебры – Parallel Basic Linear Algebra Subroutines).

Библиотека PBLAS включает библиотеку подпрограмм BLAS (Basic Linear Algebra Subroutines) для осуществления базовых операций линейной алгебры на серийных компьютерах и библиотеку BLACS (Basic Linear Algebra Communication Subroutines) для организации параллельных вычислений. Библиотека PBLAS является основной компонентой библиотеки ScaLAPACK (Scalable LAPACK), предназначенной для решения отдельной задачи линейной алгебры, например, решения системы линейных алгебраических уравнений, нахождения собственных значений вещественной симметричной матрицы, *LU*-разложения матрицы, приведения вещественной симметричной матрицы к трехдиагональному виду [6,7].

BLAS включает в себя высокоэффективные подпрограммы, написанные на языках Fortran и C, которые являются фундаментальными блоками для реализации базовых векторных и матричных операций. Эти подпрограммы учитывают иерархию памяти современных процессоров и делятся на три уровня. В библиотеке BLAS первого уровня (BLAS1) собраны подпрограммы для операций над векторами или

парами векторов. Они выполняют $O(n)$ арифметических операций, результатом которых является либо вектор, либо число. Примерами подпрограмм BLAS1 являются: операция saxpy, скалярное произведение, вычисление нормы вектора (табл. 2.1).

Таблица 2.1

Примеры подпрограмм библиотеки BLAS

Уровень	Кол-во операций	Примеры	Функция
BLAS1	$O(n)$	*Saxpy* *Sdot* *Snrm2*	Скаляр×вектор+вектор Скалярное произведение Евклидовая норма вектора
BLAS2	$O(n^2)$	*Sgemv* *Strsv*	Произведение матрицы на вектор Решение СЛАУ с треугольной матрицей
BLAS3	$O(n^3)$	*Sgemm* *Strsm*	Произведение матриц Решение нескольких систем с треугольными матрицами

BLAS второго уровня (BLAS2) объединяет подпрограммы для матрично-векторных операций. Эта часть библиотеки BLAS спроектирована таким образом, чтобы оптимизировать использование данных кэш-памяти и уменьшить количество обменных операций между быстрой и медленной памятью. BLAS2 совершает операции главным образом с матрицей (двумерным массивом) и вектором (или векторами), результатом которых являются матрица или вектор. Если массив имеет размерность $n \times n$, то необходимое количество операций оценивается как $O(n^2)$.

BLAS третьего уровня (BLAS3) предназначена для реализации матричных операций и учитывает все преимущества современной архитектуры компьютера, в значительной степени минимизирует в вычислительном алгоритме отношение числа операций с плавающей точкой к количеству обращений к быстрой памяти. Данные, размещенные в кэш-памяти или локальной памяти, оптимально используются за счет разбиения матриц на более мелкие блоки. Матричные операции осуществляются на основе блочного представления матриц. BLAS3 оперирует с парами или тройками матриц. В результате получается матрица. Примерами подпрограмм из библиотеки BLAS3

являются подпрограмма умножения матриц и подпрограмма решения нескольких систем с треугольными матрицами (табл. 2.1).

Поскольку библиотека BLAS включает высокоэффективные подпрограммы, может применяться в различных вычислительных системах и широко используется мировым научным сообществом, в настоящее время она является основным средством при разработке высококачественного программного обеспечения для решения задач вычислительной линейной алгебры, например пакетов LINPACK и LAPACK. Библиотека BLAS распространяется свободно (http://www.netlib.org/blas).

BLACS – библиотека для обеспечения параллельных вычислений при решении задач линейной алгебры, использующая стандарт передачи сообщений. Эта библиотека организует параллельные вычисления на одно- или двумерной логической процессорной сетке, где каждый процесс обрабатывает части матриц или векторов. BLACS включает подпрограммы для синхронного получения или отправки сообщений, чтобы обеспечить передачу матриц или их блоков с одного процесса другому, рассылку блоков матриц нескольким процессам или глобальную редукцию данных (суммирование, определение максимального или минимального значения). В этой библиотеке также есть подпрограммы для построения и модификации логической процессорной сетки. В настоящее время BLACS эффективно используется на многопроцессорных вычислительных системах, где установлены библиотеки передачи сообщений, например MPI или PVM.

3. ПРЯМЫЕ МЕТОДЫ РЕШЕНИЯ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЙ

3.1. Решение систем линейных уравнений с заполненными матрицами методом исключения Гаусса

Рассмотрим систему линейных алгебраических уравнений

$$A\vec{x} = \vec{b} \tag{3.1}$$

с невырожденной матрицей A размера $n \times n$. Будем считать матрицу A – плотной, т.е. содержащей относительно небольшое число нулевых элементов.

Одним из прямых методов решения линейных систем является метод исключения Гаусса. Идея этого метода состоит в том, что матрица A сначала упрощается – приводится эквивалентными преобразованиями к треугольной или диагональной, а затем решается система с упрощенной матрицей. Наиболее известной формой исключения Гаусса является та, в которой расширенная матрица системы линейных уравнений приводится к верхнетреугольному виду путём вычитания одних уравнений, умноженных на подходящие числа, из других уравнений. Полученная треугольная система решается с помощью обратной подстановки.

Применение метода исключения Гаусса основывается на *LU-теореме.* Пусть дана квадратная матрица A порядка n, и пусть A_k – главный минор матрицы, составленный из первых k строк и столбцов. Если $\det A_k \neq 0$ для $k = 1,2,...,n$, то тогда существуют единственная нижняя треугольная матрица L с единичными элементами на главной диагонали и единственная верхняя треугольная матрица U такие, что $A = LU$.

В этом случае, используя полученное разложение, система линейных уравнений запишется следующим образом:

$$LU\vec{x} = \vec{b}\,. \tag{3.2}$$

Тогда исходная система (3.1) сводится к последовательному решению двух систем:

- решая систему с нижней треугольной матрицей $L\vec{y} = \vec{b}$ прямой подстановкой, получим вектор $\vec{y}$;

- затем, решая систему с верхней треугольной матрицей $U\vec{x} = \vec{y}$ обратной подстановкой, находим решение $\vec{x}$ исходной системы уравнений (3.1).

Рассмотрим вначале LU-разложение, поскольку проведение этой операции требует значительных временных затрат. Ниже представлен псевдокод LU-факторизации, где значения диагональных и наддиагональных коэффициентов матрицы U в ходе вычислительного процесса переписываются в соответствующие элементы матрицы A. Пусть $a_{ii} \neq 0, i = 1,...,n$.

```
do k=1,n-1
   do i=k+1,n
     lik = aik/akk
   end do
   do j=k+1,n
      do i=k+1,n
         aij = aij – lik*akj
      end do
   end do
end do
```

В цикле j производится вычитание k-й строки матрицы A, умноженной на соответствующее число, из расположенных ниже строк. Правая часть $\vec{b}$ системы (3.1) также может быть добавлена к матрице A ($n+1$ столбец) и обрабатываться в ходе приведения к треугольному виду. В общем случае может потребоваться перестановка строк для обеспечения корректности и вычислительной устойчивости LU-разложения (выбор главного элемента в случае, когда $a_{ii} = 0$ или $a_{ii} \simeq 0$).

В методе исключения Гаусса выполняется около $n^3/3$ парных операций сложения и умножения, а также $n^2/2$ делений. Пренебрегая членом меньшего порядка, получим для последовательных вычислений

$$T_1 \approx \left(t_{mult} + t_{add}\right) n^3 / 3 .$$

LU-разложение (метод исключения Гаусса) представляется в виде тройных вложенных циклов, в которых рассчитанные элементы матрицы *U* переприсваивают элементам исходной матрицы A:

```
do _____________
  do _____________
    do _____________
       aij = aij – (aik/akk)/akj
    end do
  end do
end do
```

Индексы i, j, k в циклах могут быть взяты в любом порядке. Всего возможно 3! = 6 вариантов задания порядка следования циклов, как и для рассмотренного в предыдущей главе в п.2.3 умножения квадратных матриц. Шесть различных форм следования *ijk*-циклов отличаются способом доступа к элементам матриц A и $L = \left[l_{ik}\right]$, который может существенно повлиять на производительность алгоритма в зависимости от архитектуры компьютера. Формы *jki* и *kji* столбцово-ориентированы: они обращаются к столбцам матриц A и L. В формах *kij* и *ikj* происходят обращения к строкам матрицы A и лишь к отдельным элементам L. Для оставшихся форм (*ijk* и *jik*) характерен смешанный доступ – к столбцам в A и к строкам в L. По-видимому, наиболее оптимальными для параллельных реализаций являются формы *kij* и *kji*, которые отличаются только в форме доступа к элементам матрицы A – по строкам или по столбцам соответственно.

Применим общий подход построения параллельных процедур решения задач для алгоритма *LU*-разложения.

1. *Декомпозиция*. Выберем в качестве отдельных блоков (фундаментальных подзадач) алгоритма метода исключения Гаусса следующие: для $i, j = 1,...,n$ каждая фундаментальная задача (i, j) по значению a_{ij} вычисляет и запоминает значения

$$\begin{cases} u_{ij}, i \le j \\ l_{ij}, i > j \end{cases}.$$

Все такие фундаментальные задачи образуют двумерный массив n^2 подзадач (рис. 3.1). Нулевые элементы матриц $U = [u_{ij}]$ и L, а также единичные элементы на главной диагонали матрицы L не рассчитываются.

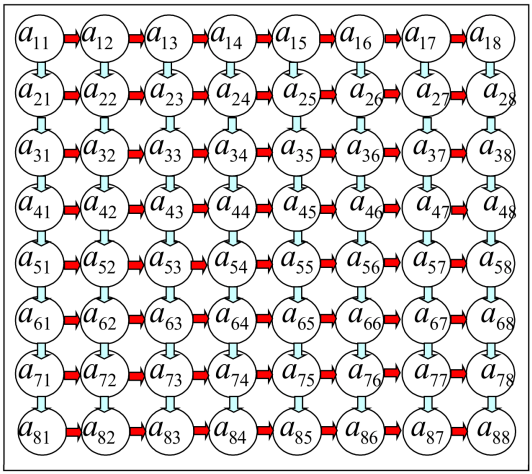


Рис. 3.1. Фундаментальные подзадачи и их коммуникационная схема (n = 8)

2. Организация коммуникаций. Необходимые коммуникации для связи фундаментальных подзадач для обеспечения всего цикла вычислений представляет следующий псевдокод для задачи (i, j):

```
do k=1,min(i,j)-1
    получить a_kj
        получить l_ik
    a_ij = a_ij – l_ik*a_kj
end do
if i≤j then
    разослать a_ij подзадачам (k,j), k=i+1,…,n
else
     получить a_jj
    l_ij = a_ij/a_jj
```

разослать l_{ij} подзадачам (i,k), k=j+1,…,n
end if

Из рис. 3.1 видно, что каждая фундаментальная подзадача (i, j) связана с подзадачами $(1, j),...,(n, j)$ и $(i,1),...,(i,n)$.

3. *Укрупнение*. Для имеющего $n \times n$ массива фундаментальных подзадач могут быть использованы следующие стратегии получения p ($p << n^2$) составных подзадач:

- одномерное укрупнение: объединяются *n/p* строк (или столбцов) двумерного массива фундаментальных подзадач;
- двумерное укрупнение: одна составная подзадача включает в себя $(n / \sqrt{p}) \times (n / \sqrt{p})$ фундаментальных подзадач.

4. *Распределение* составных подзадач *по процессорным элементам* производится с учетом архитектуры многопроцессорной вычислительной системы.

Сначала рассмотрим одномерное укрупнение, когда составная подзадача объединяет *n/p* строк массива фундаментальных подзадач (рис. 3.2).

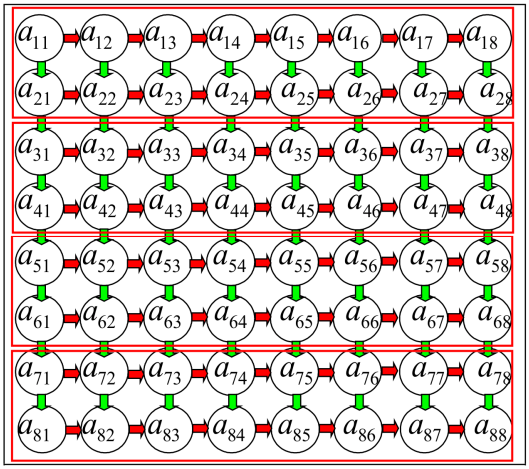


Рис.3.2. Одномерное укрупнение строк массива фундаментальных подзадач

В этом случае нет необходимости рассылать множители l_{ij} (коэффициенты матрицы L) по строкам, так как любая строка заданной матрицы целиком принадлежит одной составной подзадаче. Также следует заметить, что при данном способе укрупнения (когда строки последовательно по своему номеру распределяются составным подзадачам) нет никакого параллелизма при расчете новых значений любой заданной строки. После пересчета элементов в строке необходима рассылка полученных значений, чтобы обеспечить вычисления для каждой строки нижней составной подзадачи (рис. 3.2). Рассмотренная параллельная процедура LU-разложения может быть представлена следующим псевдокодом:

```
do k = 1,n-1
  if k ∈ myrows then
     переслать {a_kj: k≤j≤n} другим
     составным подзадачам
  else
     получить {a_kj: k≤j≤n}
  end if
  do i ∈ myrows, i>k
     l_ik = a_ik/a_kk
  end do
  do j = k+1,n
    do i ∈ myrows, i>k
       a_ij = a_ij – l_ik*a_kj
    end do
   end do
end do
```

Здесь ***myrows*** – множество индексных значений номеров строк, распределенных одной составной подзадаче.

В этом алгоритме на каждом шаге k каждой задаче требуется около $(n-k)^2 / p$ арифметических операций. Поэтому без учета коммуникационных затрат получим

$$T_p^{comp} \approx \left(t_{mult} + t_{add}\right) \sum_{k=1}^{n-1} \left(n-k\right)^2 / p \approx \left(t_{mult} + t_{add}\right) n^3 / 3p \ .$$

Количество данных, которые необходимо переслать на каждом шаге, около $(n-k)$. Тогда

$$T_p^{comm} \approx \sum_{k=1}^{n-1} t_{comm}\left(n-k\right) \approx t_{comm} n^2 / 2 .$$

Здесь t_{comm} – время на передачу одного числа. Общее время для вычисления *LU*-разложения на *p*-процессорной вычислительной системе будет

$$T_p = T_p^{comp} + T_p^{comm} \approx \left(t_{mult} + t_{add}\right) n^3 / 3p + t_{comm} n^2 / 2$$

и

$$S_p = \frac{T_1}{T_p} \approx \frac{\left(t_{mult} + t_{add}\right) n^3 / 3}{\left(t_{mult} + t_{add}\right) n^3 / 3p + t_{comm} n^2 / 2} = \frac{p}{1 + \dfrac{3pt_{comm}}{2n\left(t_{mult} + t_{add}\right)}} . \quad (3.3)$$

Чтобы уяснить особенности построенной процедуры для параллельного *LU*-разложения, рассмотрим случай, когда $p = n$. Тогда i-я строка матрицы A хранится в i-м процессорном элементе. На первом шаге вычислительного процесса первая строка рассылается на все вычислительные узлы, а процессорные элементы с номерами $2,3,...,n$ выполняют параллельные вычисления:

$$l_{i1} = a_{i1} / a_{11} ,$$
$$a'_{ij} = a_{ij} - l_{i1} a_{1j} , \ j = 2,...,n .$$

На втором шаге процессорный элемент с номером 2 рассылает вторую строку преобразованной матрицы A процессорным элементам $3,4,...,n$, на которых вновь проводятся параллельные вычисления:

$$l_{i2} = a'_{i2} / a'_{22} ,$$
$$a''_{ij} = a'_{ij} - l_{i2} a'_{2j} , \ j = 3,...,n .$$

Этот процесс повторяется для следующей строки преобразованной матрицы, однако число активных процессоров на каждом шаге уменьшается на единицу, что существенно снижает общую эффективность параллельного алгоритма. Другим недостатком представ-

ленного подхода является необходимость передачи значительного объема данных на каждом шаге.

Таким образом, для одномерного строчно-ориентированного укрупнения в алгоритме LU-разложения характерны следующие особенности:

1. Каждая составная подзадача считается выполненной, как только заканчивается обработка последней строки распределенной на процессорный элемент части матрицы A. Если составная подзадача объединяет строки матрицы A с последовательной нумерацией, то обрабатывающий строку процессорный элемент завершает свою работу задолго до окончания общего вычислительного процесса.

2. Обработка строк в порядке увеличения их номера i требует существенно меньшего объема вычислительной работы (по сравнению, например, с обработкой строки с номером 1), что ведет к неравномерной загрузке процессоров.

3. Балансировка загрузки процессоров может быть улучшена, если строки матрицы A для составных подзадач объединять циклически, т.е. строка i назначается процессорному элементу с номером $i \bmod p$, $i = 1,...,n$ (рис. 3.3).

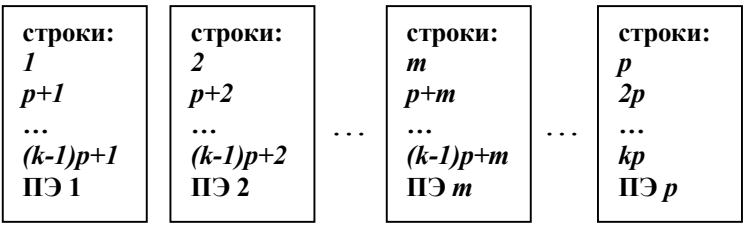


Рис. 3.3. Циклическая строчная схема распределения фундаментальных подзадач по процессорным элементам ($k = n/p$)

Проблема эффективности загрузки процессоров для этой схемы несколько теряет остроту. Так, процессорный элемент с номером 1 (**ПЭ 1**) производит вычисления при $p << n$ практически до окончания процесса факторизации матрицы A. Тем не менее некоторая неравномерность загрузки процессоров при такой схеме сохраняется: после первого же шага параллельных вычислений **ПЭ 1** придется обрабатывать на одну строку меньше, чем остальным. Кроме того, ко-

личество необходимых арифметических операций для обработки одной строки зависит от ее номера.

Также можно использовать другие способы распределения данных по процессорным элементам. Например, циклическое размещение блоков матрицы A или применение слоистой схемы с отражениями (рис. 3.4). Для этой схемы первые p строк матрицы A распределяются между p процессорными элементами в естественном порядке, следующие p строк размещаются в обратном порядке и т.д.

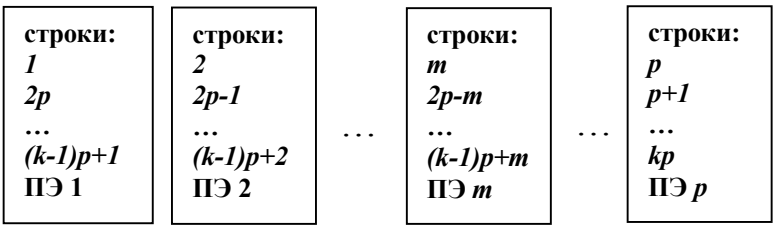


Рис. 3.4. Слоистая строчная схема с отражениями, $k = n/p$ – нечетное

4. Производительность параллельных вычислений может быть увеличена за счет совмещения коммуникационных операций и вычислений. Рассмотрим, как это можно выполнить для циклической схемы распределения данных (рис. 3.3). Пусть процесс обработки строк ведется в порядке увеличения их номера. Тогда на первом шаге каждый процессорный элемент, получив от **ПЭ 1** первую строку, сначала вычисляет множители l_{i1}, а затем модифицирует соответствующие строки матрицы A. На втором шаге процессорный элемент, хранящий вторую строку преобразованной матрицы, должен переслать ее на остальные вычислительные узлы. Здесь возникает задержка вычислительного процесса. Выходом из этой ситуации является немедленная рассылка обработанной второй строки, пока все процессоры будут заняты модификациями первого шага. Подобным образом можно организовать параллельные вычисления и на других шагах: на m-ом шаге (m+1)-я строка рассылается сразу после того, как закончится ее обработка. Такая стратегия осуществления параллельных вычислений называется ***опережающей рассылкой***. Ее эффективность в значительной степени определяется топологией межпроцессорных соединений МВС с распределенной памятью. Страте-

гию опережающей рассылки можно использовать и для систем с разделяемой памятью. Однако в силу особенностей архитектуры МВС в данном случае эта стратегия носит название ***опережающего вычисления***, поскольку меняется порядок действий. Как только закончится модификация ($m+1$)-й строки, она маркируется признаком «готова». Тогда другие процессоры, завершив свою работу на m-ом шаге, могут сразу приступать к следующему ($m+1$)-му шагу. Маркировка дает возможность осуществлять необходимую синхронизацию работы процессоров без неоправданных задержек. Часто опережающую рассылку и опережающее вычисление называют ***конвейеризацией***.

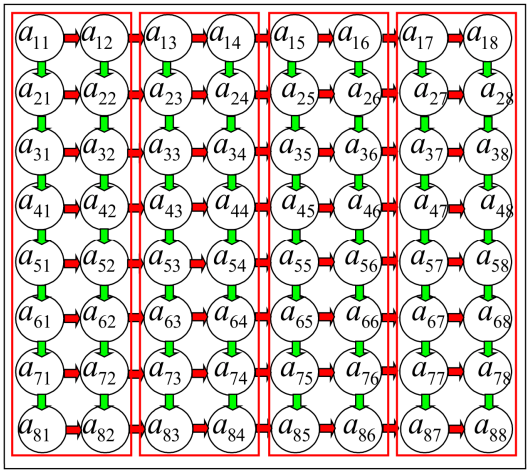


Рис. 3.5. Столбцово-ориентированное укрупнение фундаментальных подзадач для *LU*-разложения

Альтернативой строкового укрупнения фундаментальных подзадач является их объединение по столбцам (рис. 3.5). Заметим, что в этом случае нет необходимости в пересылке строк матрицы A, поскольку все необходимые для вычислений данные принадлежат одной составной подзадаче или процессорному элементу. Однако при таком способе распределения данных нет явного параллелизма при вычислении множителей l_{ij} и модификации элементов в столбцах матрицы. Также очевидна необходимость организации рассылки

множителей l_{ij} подзадачам, расположенным справа (рис. 3.5), для модификации элементов матрицы A.

Запишем псевдокод параллельной процедуры LU-разложения при распределении матрицы A по столбцам в процессорных элементах.

```
do k = 1,n-1
  if k ∈ mycols then
      do i = k+1,n
        l_ik = a_ik/a_kk
      end do
    переслать {l_ik: k<i≤n} другим
    составным подзадачам
  else
      получить {l_ik: k<i≤n}
  end if
  do j ∈ mycols, j>k
    do i = k+1,n
      a_ij = a_ij – l_ik*a_kj
    end do
  end do
end do
```

Здесь ***mycols*** – множество индексных значений номеров столбцов, распределенных одной составной подзадаче (процессорному элементу). Параметры ускорения и эффективности данного алгоритма весьма похожи на соответствующие выражения для строчно-ориентированного распределения данных (3.3).

Отметим характерные особенности параллельного вычислительного процесса LU-факторизации при использовании одномерного укрупнения фундаментальных подзадач:

1. Каждая составная подзадача (укрупненный блок) считается выполненной, как только закончится обработка ее последнего столбца. Поэтому, если процессорный элемент обрабатывает столбцы с последовательной нумерацией, он может оказаться без вычислительной работы задолго до окончания всего вычислительного процесса.

2. Расчет множителей и модификация столбцовых элементов матрицы требуют значительно меньшего объема вычислений при увеличении номера столбца. Для повышения согласованности работы процессоров и балансировки загрузки можно использовать циклическое

распределение столбцов матрицы A по процессорным элементам: например, столбцы матрицы $j = 1,...,n$ размещать на процессорные элементы, номер которых определяется как $j \bmod p$. Кроме того, могут применяться другие способы распределения данных – блочно-циклический или слоистая столбцовая схема с отражениями.

3. Производительность построенной параллельной процедуры может быть улучшена за счет совмещения обменных операций с вычислениями. На m-ом шаге каждый процессор выполняет модификацию своей части необработанной матрицы, прежде чем осуществится переход к следующему $(m+1)$-му шагу алгоритма. Однако процессорный элемент, на котором размещается $m+1$ столбец матрицы, может провести вычисления множителей для шага $m+1$ и их рассылку на другие ПЭ перед завершением всех модификаций элементов матрицы на m-ом шаге. Стратегия опережающей рассылки позволяет процессорам начинать на каждом шаге вычислительную работу раньше по сравнению с обычным подходом, когда они ожидали получение необходимых для вычисления данных от других ПЭ.

Другим способом укрупнения является объединение $n/\sqrt{p} \times n/\sqrt{p}$ фундаментальных подзадач в один блок (рис. 3.6).

В этом случае алгоритм параллельной *LU*-факторизации совмещает свойства рассмотренных выше одномерных схем укрупнения. В частности, требуется горизонтальная и вертикальная рассылка по блокам строк матрицы и столбцов множителей l_{ij} соответственно. Итоговый алгоритм является блочной версией мелкозернистого алгоритма.

Параллельный псевдокод алгоритма *LU*-разложения для двумерного укрупнения можно представить следующим образом:

```
do k = 1,n-1
  if k ∈ myrows then
      разослать {a_kj: j ∈ mycols, j>k} другим
      подзадачам в столбце этой подзадачи
  else
      получить {a_kj: j ∈ mycols, j>k}
  end if
  if k ∈ mycols then
      do i ∈ myrows, i>k
```

```
            lik = aik/akk
        end do
      разослать {lik: i ∈ myrows, i>k} другим
       составным подзадачам в строке этой
         подзадачи
    else
          получить {lik: i ∈ myrows, i>k}
    end if
    do j ∈ mycols, j>k
      do i = myrows, i>k
          aij = aij – lik*akj
      end do
     end do
end do
```

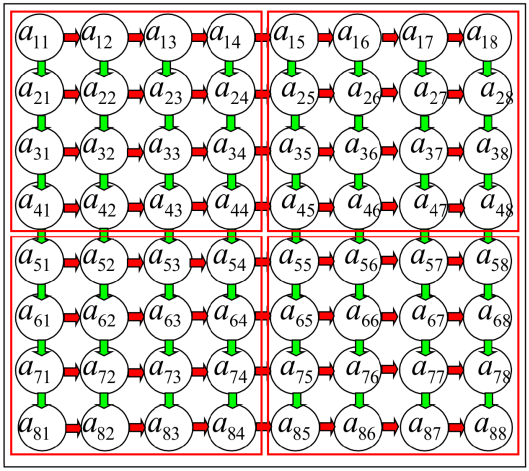


Рис. 3.6. Двумерное укрупнение мелкозернистых подзадач

На каждом шаге k количество операций, необходимых для модификации элементов матрицы каждой составной подзадачи, определяется как $(n-k)^2 / p$. Тогда общее время на проведение параллельных вычислений без учета обменных операций составит

$$T_{comp} \approx \left(t_{mult} + t_{add}\right) \sum_{k=1}^{n-1} \left(n-k\right)^2 / p \approx \left(t_{mult} + t_{add}\right) n^3 / p .$$

Количество рассылок данных на шаге k вдоль каждой строки или столбца составных подзадач (рис. 3.6) около $(n-k)/\sqrt{p}$. Тогда для двумерной сетки общее время на обменные операции можно записать как $T_{comm} \approx \sum_{k=1}^{n-1} 2\left(t_{comm}(n-k)/\sqrt{p}\right) \approx t_{comm} n^2/\sqrt{p}$.

Здесь допускается возможность частичного перекрытия рассылок на последовательных шагах параллельной процедуры.
Общее время оценивается

$$T_p \approx \left(t_{mult} + t_{add}\right) n^3/(3p) + t_{comm} n^2/\sqrt{p}$$

и

$$S_p \approx \frac{\left(t_{mult} + t_{add}\right) n^3/3}{\left(t_{mult} + t_{add}\right) n^3/(3p) + t_{comm} n^2/\sqrt{p}} = \\ = \frac{p}{1 + \dfrac{3t_{comm}\sqrt{p}}{\left(t_{mult} + t_{add}\right) n}}. \qquad (3.4)$$

Перечислим некоторые особенности параллельной реализации LU-разложения при двумерном укрупнении мелкозернистых подзадач:

1. Каждая составная подзадача считается выполненной, как только заканчивается модификация последней строки и последнего столбца блока матрицы A, распределенного этой подзадаче (процессорному элементу). В случае последовательного (по номерам) размещения строк и столбцов первый процессорный элемент заканчивает свою работу задолго до окончания всего вычислительного процесса.

2. При вычислении множителей и модификации элементов матриц все меньшее количество арифметических операций необходимо при увеличении номеров строк и столбцов для последовательного распределения элементов матриц по подзадачам. Согласованность работы процессоров и балансировка загрузки процессорных элементов может быть улучшена при использовании циклического способа распределения данных, когда элемент матрицы $a_{ij}, i, j = 1,...,n$ присваивается составной подзадаче с номером ($i \bmod \sqrt{p}, j \bmod \sqrt{p}$, рис. 3.6).

Также могут применяться и другие способы распределения: блочно-циклический и схема с отражением.

3. Производительность этого алгоритма может быть повышена за счет совмещения вычислений и межпроцессорных обменных операций путем применения стратегии опережающей рассылки.

Частичный выбор главного элемента

Чтобы обеспечить корректность и устойчивость *LU*-разложения, часто применяется схема частичного выбора главного элемента, в результате которой в непреобразованной части матрицы A производится перестановка строк таким образом, чтобы в ведущем столбце находился наибольший по модулю элемент матрицы. При такой стратегии множители l_{ik} по абсолютной величине не будут превышать единицы, что способствует уменьшению влияния ошибки округления на конечный результат.

Использование схемы частичного выбора главного элемента для обеспечения устойчивости процесса усложняет параллельную реализацию исключения Гаусса и оказывает значительное влияние на производительность параллельных вычислений. При одномерной столбцовой декомпозиции поиск главного элемента не требует обмена данными между процессорными элементами – он ведется каким-то одним процессором. Однако в этом случае остальные процессоры простаивают. Выход из такой ситуации может быть обеспечен путем опережающего вычисления и рассылки. На k -м шаге поиск главного элемента в k +1-м столбце сразу же начинается после модификации этого столбца. Как только будет найдена ведущая строка, эта информация сразу же передается другим процессорам, которые могут параллельно выполнить перестановку строк.

В заключение заметим, что систему линейных алгебраических уравнений $A\vec{x} = \vec{b}$ можно решить, пользуясь формулами Крамера ($\det A = \Delta \neq 0$)

$$x_i = \Delta_i / \Delta \ , \ i = 1, 2, ..., n, \qquad (3.5)$$

где Δ_i – определитель, получающийся из матрицы A заменой i-го столбца столбцом правых частей системы. Непосредственное вычисление определителя Δ матрицы A по формулам линейной алгебры требует $O(n!)$ операций, что при $n \gg 1$ не экономично. Поэтому для

вычисления Δ можно воспользоваться описанным выше алгоритмом прямого хода метода Гаусса (не выполняя действий со столбцом правой части системы), в результате которого будет получена верхнетреугольная матрица $U = [u_{i,j}]_1^n$ за $O(n^3)$ операций, определитель которой по LU-теореме равен определителю матрицы Δ. Все определители из (3.5) можно вычислить за $O(n^4)$ операций. Отметим, что метод Гаусса можно применить и для нахождения обратной матрицы, учитывая, что $A^{-1}A = E$.

Таким образом, используя описанные выше параллельные алгоритмы метода исключения Гаусса, можно вычислять определители и находить элементы обратной матрицы как решение n систем с различными правыми частями.

3.2. Решение систем с треугольными матрицами

После выполнения LU-разложения (3.2) необходимо решать системы линейных уравнений с треугольными матрицами: $L\vec{y} = \vec{b}$ и $U\vec{x} = \vec{y}$. Заметим, что большинство прямых методов решения линейных систем с заполненными матрицами приводят матрицу A к треугольному виду, а затем ищется решение полученной эквивалентной системы с треугольной матрицей. Кроме того, системы с треугольными матрицами часто используются для предобусловливания в итерационных методах решения СЛАУ.

Для системы с нижней треугольной матрицей $L\vec{x} = \vec{b}$ решение может быть получено в результате прямой подстановки:

$$x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii}, i = 1,...,n . \qquad (3.6)$$

Псевдокод этого алгоритма можно записать следующим образом:

```
do i = 1,n
  x_i = b_i/l_ii
  do j = i+1,n
       b_j = b_j – l_ji * x_i
  end do
end do
```

При решении системы с верхней треугольной матрицей $U\vec{x} = \vec{b}$ требуется применять обратную подстановку:

$$x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}, i = n,...,1. \qquad (3.7)$$

В этом случае псевдокод имеет вид

```
do i = n,1
  x_i =  b_i/u_ii
  do j = 1,i-1
     b_j = b_j – u_ji * x_i
  end do
end do
```

Заметим, что представленный выше псевдокод для решения системы $U\vec{x} = \vec{b}$ полезен, если матрица U хранится в памяти компьютера по столбцам. Этот алгоритм называется *столбцовым алгоритмом* (или *алгоритмом векторных сумм*). Альтернативный алгоритм, который носит название *алгоритма скалярных произведений*, имеет преимущество, когда матрица U хранится по строкам. Он задает скалярные произведения с длинами векторов, меняющимися от 1 до $n-1$, и n делений чисел.

```
do i = n,1
  do j = i+1,n
     b_i = b_i – u_ij * x_j
  end do
  x_i =  b_i/u_ii
end do
```

Анализируя эти два алгоритма, можно отметить, что в столбцовом алгоритме обратной подстановки производится немедленное (а не отложенное) определение значения i-й неизвестной. Однако какой из этих алгоритмов следует выбрать, диктуется, главным образом, способом хранения матрицы U, если он был определен в процессе LU-разложения.

Подсчитаем количество арифметических операций сложения или умножения для получения решения по (3.6) или (3.7):

$$1+2+\ldots+\ n-1=n(n-1)/2\approx n^2\ /2\ .$$

Пренебрегая временем выполнения n операций деления в (3.6) или (3.7), можно записать

$$T_1\approx\left(t_{add}+t_{mult}\right)n^2\ /2\ . \qquad (3.8)$$

Далее рассмотрим только нижнетреугольную систему $L\vec{x}=\vec{b}$, так как решение верхнетреугольной системы организуется аналогичным образом.

Разберем, как строятся параллельные алгоритмы для решения систем с нижнетреугольными матрицами.

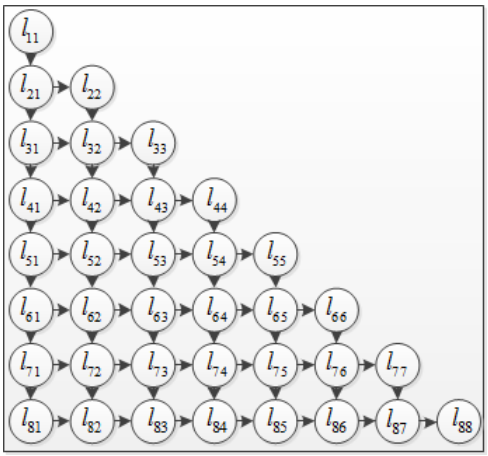


Рис. 3.7. Декомпозиция и проектирование коммуникаций для алгоритма решения системы с треугольной матрицей

На этапе декомпозиции в качестве фундаментальных подзадач будем выбирать следующие (см. рис. 3.7):

- для $i=2,...,n;\ j=1,...,i-1$ (i,j)-я подзадача запоминает l_{ij}, получает x_j и вычисляет произведение $l_{ij}x_j$;

- для $i = 1,...,n$ (i,i)-я подзадача хранит значения l_{ii} и b_i, собирает произведения $l_{ij}x_j$, вычисляет сумму $t_i = \sum_{j=1}^{i-1} l_{ij}x_j$ и вычисляет и запоминает $x_i = (b_i - t_i)/l_{ii}$.

Таким образом, декомпозиция дает двумерный треугольный массив $n(n+1)/2$ мелкозернистых подзадач.

На этапе проектирования коммуникаций (определения связанности фундаментальных подзадач) для данного алгоритма и выбранной схемы декомпозиции можно заметить, что:

- для $j = 1,...,n-1$ (j,j)-я подзадача одновременно передает значение x_j подзадачам (i,j), $i = j+1,...,n$;

- далее производится пересылка произведений $l_{ij}x_j$ от подзадач (i,j), $j = 1,...,i-1$, подзадаче (i,i).

При таком способе выбора фундаментальных мелкозернистых подзадач параллельная программа для рассматриваемого алгоритма примет вид:

Для (i,j)-й подзадачи $(i \geq j)$

```
if i=j then
    if i>1 then
        получить{t} от подзадач (i,j), j=1,…,i-1 (t-сумма ti)  else
        t=0
    end
    xi=(bi-t)/lii
    разослать {xi} подзадачам (k,i), k=i+1,…,n
else
    получить {xj}
    t=l(i,j)*x(j)
    переслать {t} для суммирования подзадаче (i,i)
end if
```

Согласно этому фрагменту параллельной программы сначала должно быть найдено значение неизвестного x_1, т.е. решена подзадача (1,1). На следующем шаге это значение пересылается подзадачам (2,1) и (2,2); одновременно вычисляются произведения; полученные каждой подзадачей значения t затем пересылаются подзада-

че (2,2), где рассчитывается значение x_2. Далее процесс продолжается, вычисляется следующее неизвестное, и видно, что с увеличением индекса неизвестных все больше вычислений может проводиться одновременно, однако также увеличивается и количество коммуникаций между подзадачами. Чтобы уменьшить затраты на передачу значений между подзадачами, необходимо провести объединение мелкозернистых подзадач.

Укрупнение для имеющегося двумерного массива фундаментальных мелкозернистых подзадач может быть выполнено следующими способами:

- одномерным укрупнением n / p строк (или столбцов);
- двумерным укрупнением $n / \sqrt{p} \times n / \sqrt{p}$ подзадач.

В итоге получается p крупнозернистых подзадач.

Планирование вычислений осуществляется путем назначения укрупненных подзадач соответствующим процессорным элементам.

Рассмотрим одномерное укрупнение по строкам мелкозернистых подзадач (рис. 3.8).

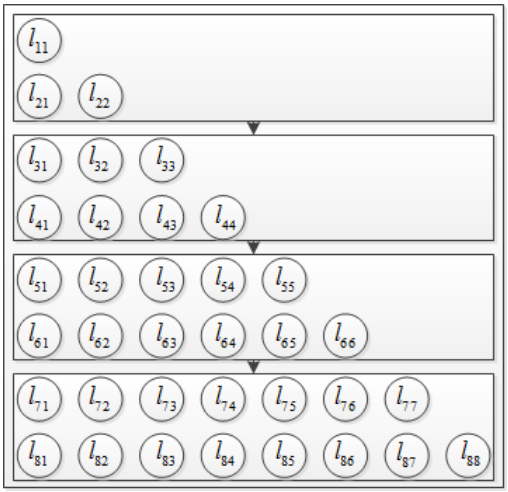
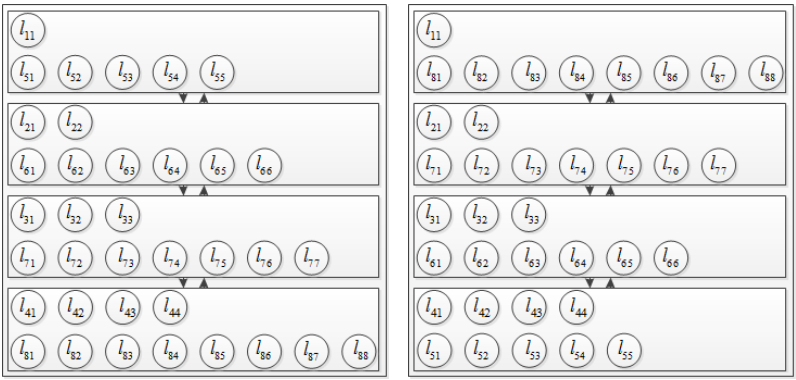


Рис. 3.8. Одномерное строковое укрупнение мелкозернистых подзадач

Видно, что такой способ укрупнения не требует передачи данных для вычисления суммы значений t по строкам, поскольку строка матрицы мелкозернистых подзадач целиком принадлежит укрупненной подзадаче. Заметим, что нет никакого параллелизма при вычислении этих сумм. Кроме того, передача вычисленных значений неизвестных другим укрупненным подзадачам все еще необходима.

Псевдокод параллельной программы можно записать следующим образом:

```
do j=1,n
    if j ∈ myrows then
        xj=bj/ljj
        разослать {xj} остальным подзадачам
    else
        получить {xj}
    end if
    do i=myrows, i>j
        bi = bi – lij*xj
    end do
end do
```



(а) (б)

Рис. 3.9. Циклическая схема (а) и схема с отражениями (б) распределения строк мелкозернистых подзадач

В соответствии с принятой схемой укрупнения можно отметить следующие особенности полученной параллельной программы:

1. Выполнение каждой укрупненной подзадачи заканчивается после решения последнего уравнения, принадлежащего этой подзадаче.

2. При последовательном распределении уравнений по блокам укрупненная подзадача может быть выполнена задолго до того, как будет закончена вся вычислительная работа по решению треугольных систем.

3. Расчет скалярных произведений для каждой строки мелкозернистых подзадач требует большего количества вычислений при увеличении ее номера.

4. Согласованность работы процессоров и балансировка их загрузки может быть улучшена за счет объединения строк в блоки в циклическом порядке, когда i-я строка фундаментальных подзадач назначается укрупненной подзадаче с номером $i \bmod p$ (рис. 3.9,а).

5. Для решения проблемы равномерной загрузки процессоров при укрупнении подзадач также могут быть использованы блочный циклический способ объединения строк или схема с отражениями (рис. 3.9,б).

Подсчитаем временные затраты параллельного алгоритма. Время, необходимое для проведения вычислений по рассмотренному алгоритму, оценим как

$$T_p^{comp} \approx n\left(t_{div} + \left(t_{add} + t_{mult}\right)\frac{n}{2p} \right),$$

здесь t_{div} – время операции деления двух чисел с плавающей точкой, а время, затрачиваемое на рассылку рассчитанных значений неизвестных, представим в виде

$$T_p^{comm} \approx t_{comm}\frac{n\left(p-1\right)}{p}.$$

В итоге

$$T_p \approx \left(t_{add} + t_{mult}\right)\frac{n^2}{2p} + t_{comm}\frac{n\left(p-1\right)}{p}. \qquad (3.9)$$

Проведем оценку ускорения построенного алгоритма по формулам (3.8) и (3.9) в предположении, что обменные операции между узлами перекрываются, т.е. происходят одновременно:

$$S_p = \frac{T_1}{T_p} \approx \frac{p}{1 + \dfrac{2t_{comm}(p-1)}{(t_{add} + t_{mult})n}}. \qquad (3.10)$$

Из (3.10) следует, что повышение эффективности построенной параллельной программы может быть достигнуто при уменьшении доли временных затрат на обеспечение пересылки данных между вычислительными узлами. Одним из путей решения этой проблемы является использование стратегии опережающей рассылки, т.е. когда рассылка данных совмещается с вычислениями. Для рассматриваемой задачи на j-м шаге (см. представленный выше алгоритм) крупнозернистая подзадача, включающая строку (уравнение) $j+1$, может рассчитать значение x_{j+1} и разослать его другим подзадачам перед осуществлением окончательных вычислений с использованием x_j. Эффект от опережающей рассылки будет более значительным, если увеличивается количество перекрытий последовательных шагов алгоритма. Такое перекрытие существенно зависит от топологии многопроцессорной вычислительной системы и от способа распределения строк мелкозернистых задач по процессорам.

Например, при циклической схеме распределения (рис. 3.9,а) на МВС с топологией «кольцо» имеет место практически полное перекрытие обменов, в то время как топология «гиперкуб» позволяет делать существенно меньшее количество перекрытий.

Рассмотрим одномерное укрупнение фундаментальных мелкозернистых подзадач по столбцам. Для такого объединения подзадач в p блоков (рис. 3.10) характерными являются следующие особенности:

1. Нет необходимости рассылать вычисленные значения неизвестных $\{x_j\}$ по вертикали, поскольку каждый необходимый для расчетов столбец матрицы целиком принадлежит только одной укрупненной подзадаче.

2. Нет никакого параллелизма при расчете произведений $l_{ij}x_j$.

3. Для вычисления суммы произведений $\sum_j l_{ij} x_j$ потребуются пересылки между укрупненными блоками по горизонтали.

4. Каждая укрупненная подзадача начинает выполняться только тогда, когда начнет вычисляться неизвестная x_j ее первого столбца.

5. Если каждая крупнозернистая подзадача объединяет последовательные блоки столбцов, она может оставаться неактивной большую часть процесса решения задачи.

6. Количество произведений, осуществляемых с одной компонентой вектора $\vec{x}$, уменьшается с увеличением номера столбца.

7. Равномерная загрузка процессорных элементов может быть улучшена за счет назначения столбцов блокам по циклической схеме, когда столбец j распределяется укрупненному блоку с номером $j \bmod p$. Кроме того, могут использоваться другие схемы, например блочно-циклическая или схема с отражениями.

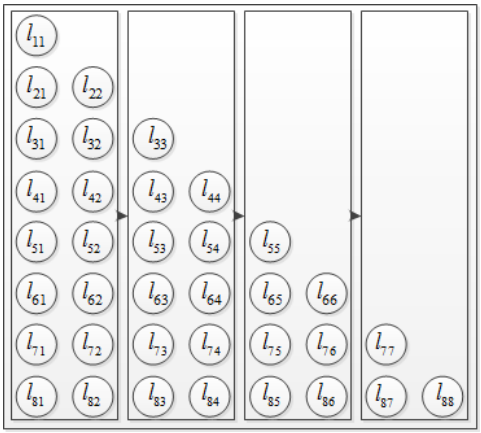


Рис. 3.10. Столбцово-ориентированная схема
распределения мелкозернистых подзадач по блокам

Оценки показывают, что для одномерного столбцового укрупнения время выполнения параллельного алгоритма, в сущности, точно такое же, как и для одномерного строкового и аналогично зависит от перекрытия последовательных шагов алгоритма. Ускорение парал-

лельного алгоритма может быть улучшено за счет применения стратегии опережающих вычислений.

```
do i=1,n
    t=0
    do j∈mycols, j<i,
        t=t+l_ij x_j
    end do
    if i∈mycols then
        получить {t} и провести суммирование
                полученных значений
        x_i=(b_i-t)/l_ii
    else
        отправить {t} для вычисления суммы
    end if
end do
```

Параллелизм рассмотренных выше алгоритмов получается из внутреннего цикла, выполнение которого распределяется по укрупненным задачам, при этом внешний цикл выполняется последовательно. Отметим, что эти параллельные алгоритмы вычисляют только по одной компоненте решения $\vec{x}$. Для повышения эффективности можно применить конвейерный способ, в котором используется явный параллелизм и для внешнего цикла, что позволяет рассчитывать несколько компонент вектора решения одновременно. Такие алгоритмы получили название волновых.

Действительно, рассмотрим одномерную схему укрупнения по столбцам (рис. 3.11). Кажется, что этот алгоритм не имеет никакого параллелизма: после того, как укрупненная подзадача, располагающая столбцом j, вычислит x_j, окончательное обновление правой части не может быть разделено с другими подзадачами для параллелизации, так как они не имеют доступа к столбцу j. Вместо того, чтобы выполнять все такие обновления немедленно, подзадача, располагающая столбцом j, могла бы рассчитать только, скажем, s компонентов обновляемого вектора и пересылать (направлять) их подзадаче, имеющей столбец $j+1$, перед продолжением работы со следующими s компонентами обновляемого вектора. По мере получения первых s компонентов обновляемого вектора подзадача, содержа-

щая столбец $j+1$, может вычислять x_{j+1}, проводить дальнейшие обновления по собственному вхождению в другие подзадачи и т.д.

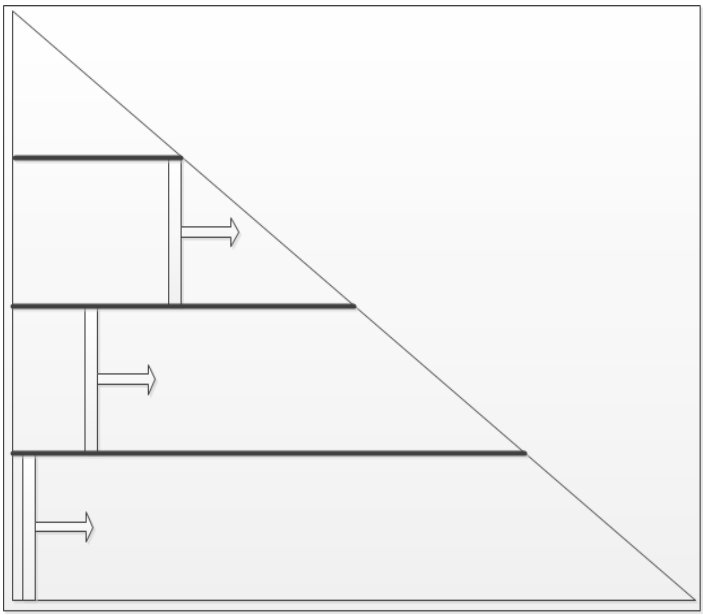


Рис. 3.11. Одномерный столбцовый волновой алгоритм

Чтобы формализовать волновой столбцовый алгоритм, введем вектор $\vec{z}$, в котором будут накапливаться обновления вектора правых частей $\vec{b}$, и определим сегмент, в котором содержится, по крайней мере, s последовательных компонентов $\vec{z}$.

```
do j∈mycols
    do k=1, #segments
        получить {segment}
        if k=1 then
            x_j=(b_j-z_j)/l_jj
            segment=segment-{z_jj}
        end if
        do z_i∈segment
            z_i=z_i+l_ij x_j
        end do
```

```
        if |segment|>0 then
            отправить {segment} подзадаче
                        со столбцом j+1
        end if
    end do
end do
```

В зависимости от размера сегмента, расположения столбца, отношения скорости обменов к скорости вычислений и т.п. все подзадачи возможно загрузить одновременной работой, если ее выполнять для различных компонентов решения. В приведенном выше псевдокоде ***Segment*** – это регулируемый параметр, который контролирует соотношение временных затрат между обменами и параллельными вычислениями. «Первый» сегмент для данного столбца сдвигается на один элемент после того, как каждая компонента решения найдена, исчезает после s шагов, затем следующий сегмент становится «первым» сегментом и т.д. К концу выполнения алгоритма остается только один сегмент, и он содержит только один элемент. Таким образом, в этом алгоритме снижается общее количество обменов.

Время, затрачиваемое на выполнение волнового столбцового алгоритма, может быть оценено следующим образом:

$$T_p \approx \left(t_{comm} + t_{add} + t_{mult}\right)\left(n^2 + np + s\left(s-1\right)p^2\right)/\left(2p\right). \qquad (3.11)$$

По мере того как длина сегмента s увеличивается, затраты на запуск коммуникационных операций уменьшаются, но растут затраты на вычисления. Обратная картина имеет место для (3.11) при уменьшении длины сегмента. Оптимальное значение длины сегмента s может быть определено из модели.

Волновой алгоритм также может применяться и в случае одномерного строкового укрупнения. В волновых алгоритмах каждый сегмент отправляется не более s раз и может проходить через ту же самую подзадачу несколько раз, в зависимости от распределения строк или столбцов.

3.3. Решение линейных систем с трёхдиагональными матрицами

Рассмотрим систему линейных алгебраических уравнений с трехдиагональной матрицей следующего вида:

$$\begin{pmatrix} b_0 & -c_0 & 0 & 0 & 0 & \dots & 0 \\ -a_1 & b_1 & -c_1 & 0 & 0 & \dots & 0 \\ 0 & -a_2 & b_2 & -c_2 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -a_{n-2} & b_{n-2} & -c_{n-2} & 0 \\ 0 & \dots & 0 & 0 & -a_{n-1} & b_{n-1} & -c_{n-1} \\ 0 & \dots & 0 & 0 & 0 & -a_n & b_n \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}. \quad (3.12)$$

Системы (3.12) образуют очень важный класс линейных алгебраических уравнений. Они часто получаются в результате разностных аппроксимаций дифференциальных краевых задач, а также при построении кубических сплайнов. Экономичными прямыми методами решения таких систем уравнений на компьютерах с последовательной архитектурой являются специальные варианты метода исключения Гаусса – метод прогонки и методы редукции.

К сожалению, многие методы, которые оказались наиболее эффективными в последовательных алгоритмах, не подходят для прямого использования на параллельных компьютерах. Примером такого метода является метод прогонки.

Метод прогонки

Рассмотрим формулы метода прогонки для последовательных вычислений. Пусть необходимо решить следующую систему линейных уравнений:

$$\begin{cases} -a_i x_{i-1} + b_i x_i - c_i x_{i+1} = f_i, i = 0,...,n; \\ a_0 = 0, c_n = 0. \end{cases} \quad (3.13)$$

Будем рассматривать только системы вида (3.13), которые имеют диагональное преобладание ($|b_i| \geq |a_i| + |c_i|, i = 0,...,n$, причем хотя бы в одном из перечисленных неравенств обязательно должно выполняться строгое неравенство).

Для решения системы (3.13) методом прогонки необходимо выполнить:

- вычисление прогоночных коэффициентов («прямой ход» метода)

$$P_0 = \frac{c_0}{b_0};\ Q_0 = \frac{f_0}{b_0};$$

$$P_i = \frac{c_i}{b_i - a_i P_{i-1}},\ Q_i = \frac{f_i + a_i Q_{i-1}}{b_i - a_i\ P_{i-1}},\ i = 1, ..., n-1; \qquad (3.14)$$

($\{P_i\}, \{Q_i\}$ – прогоночные коэффициенты)

- определение значений неизвестных («обратный ход» метода)

$$x_n = \frac{f_n + a_n\ Q_{n-1}}{b_n - a_n P_{n-1}};\quad x_i = P_i x_{i+1} + Q_i,\ i = n-1, ..., 0. \qquad (3.15)$$

Количество арифметических операций, необходимое для проведения расчетов по формулам (3.14), (3.15), определяется как $6(n-1) + 2n + 7 \approx 8n$. То есть при увеличении размера системы (3.12) количество последовательных вычислений или время счета по последовательной программе, реализующей метод прогонки, возрастет линейно.

Заметим, что формулы (3.14), (3.15) представляют не единственный способ применения метода прогонки. Например, если рассматривать вместо (3.15) рекуррентную формулу, в которой неизвестные линейной системы определяются в порядке увеличения индекса i, то получается:

- «прямой ход» – вычисление прогоночных коэффициентов $\{R_i\}, \{T_i\}$:

$$R_n = \frac{a_n}{b_n}, T_n = \frac{f_n}{b_n};$$

$$R_i = \frac{a_i}{b_i - c_i R_{i+1}}, T_i = \frac{f_i + c_i T_{i+1}}{b_i - c_i R_{i+1}}; i = n-1, ..., 1; \qquad (3.16)$$

- «обратный ход» – вычисление неизвестных

$$x_0 = \frac{f_0 + c_0 T_1}{b_0 - c_0 R_1};\ \ x_i = R_i x_{i-1} + T_i, i = 1, .., n. \qquad (3.17)$$

Из формул (3.16) и (3.17) видно, что построенный вариант метода прогонки имеет такое же количество арифметических операций, что и (3.14), (3.15), а отличается только порядком изменения индексной переменной при вычислении прогоночных коэффициентов и неизвестных. Оба рассмотренных метода решения системы линейных уравнений (3.12) являются последовательными, поскольку в них рассматриваются рекуррентные формулы – как линейные (о способах распараллеливании которых указывается в главе 1), так и нелинейные. При выборе варианта метода прогонки преимущество имеет тот, в котором первые прогоночные коэффициенты ($\{P_i\}$ или $\{R_i\}$) имеют наименьшие по абсолютной величине значения, что в результате снизит влияние ошибки округления на конечный результат.

Если применять формулы (3.14), (3.15) для вычисления прогоночных коэффициентов $\{P_i\},\{Q_i\}$ и неизвестных $\{x_i\}$ для значений индекса $i = 0,...,n/2-1$, а (3.16), (3.17) – для вычисления $\{R_i\},\{T_i\}$ и $\{x_i\}$ при $i = n,...,n/2+1$, то можно получить вариант метода прогонки, в котором вычисления на этапах прямого и обратного хода могут проводиться независимо и допускать распараллеливание построенного алгоритма.

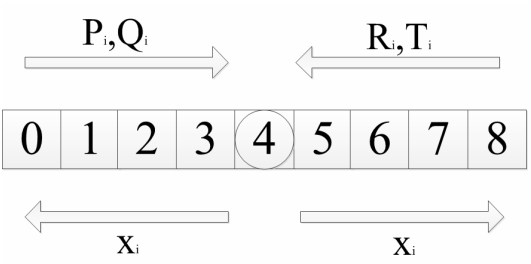


Рис. 3.12. Метод встречных прогонок, $n = 8$

Действительно (см. рис. 3.12), для своих подобластей изменения индексной переменной i одновременно могут находиться прогоночные коэффициенты $\{P_i\},\{Q_i\}$ и $\{R_i\},\{T_i\}$. И если известно значение $x_{n/2}$, то можно по формулам (3.15) и (3.17) одновременно и незави-

симо вычислить все неизвестные. Для равномерной загрузки процессоров необходимо, чтобы n было четным.

Такой вариант метода прогонки получил название «метод встречных прогонок». Заметим, что значение неизвестной $x_{n/2}$ может быть найдено после первого этапа метода встречных прогонок – определения прогоночных коэффициентов. Для этого в уравнении

$$-a_{n/2}x_{n/2-1} + b_{n/2}x_{n/2} - c_{n/2}x_{n/2+1} = f_{n/2}$$

необходимо, пользуясь формулами (3.15) и (3.17), исключить $x_{n/2\pm1}$. Тогда получим

$$x_{n/2} = \frac{f_{n/2} + a_{n/2}Q_{n/2-1} + c_{n/2}T_{n/2+1}}{b_{n/2} - a_{n/2}P_{n/2-1} - c_{n/2}R_{n/2+1}}. \qquad (3.18)$$

Идея метода встречных прогонок была обобщена в работе Н.Н. Яненко с сотрудниками на случай многопроцессорной системы и получила название параметрической прогонки. Основу предлагаемого метода составляют:

- выделение части искомых неизвестных $\{x_i\}$ в качестве параметрических (обозначим их $\{\overline{x}_\mu\}$);

- построение системы линейных уравнений относительно этих неизвестных;

- решение этой системы и определение $\{\overline{x}_\mu\}$;

- нахождение по $\{\overline{x}_\mu\}$ остальных искомых неизвестных $\{x_i\}$.

Для осуществления этой идеи система уравнений (3.12) разбивается на подсистемы, количество которых будет совпадать с количеством используемых процессорных элементов p. Причем декомпозиция по данным осуществляется таким образом, что каждый μ-й процессорный элемент обрабатывает $m+1 = n/p+1$ уравнений (см. рис. 3.13), а для неизвестных системы вводятся локальные индексы для получения однородного алгоритма:

$$x_i = x_{j+\mu m} = x_j^{(\mu)},\ i = 0,...,n;\ j = 0,...,m;\ \mu = 0,...,p-1.$$

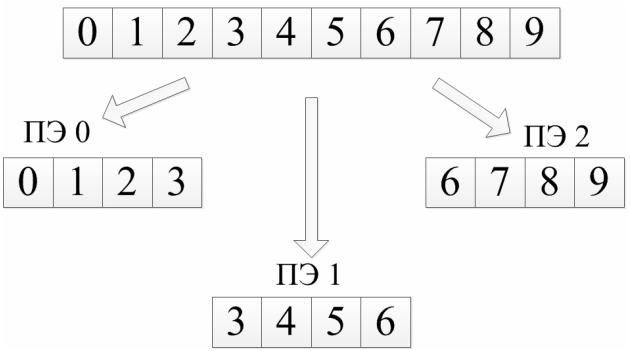


Рис. 3.13. Пример декомпозиции неизвестных между тремя процессорами

Кроме того, принимается, что

$$x_m^{(\mu-1)} = x_0^{(\mu)} = \overline{x}_\mu, \mu = 1,..., p-1; \quad \overline{x}_0 = x_0; \quad \overline{x}_p = x_n. \qquad (3.19)$$

Решение системы (3.12) будем искать в виде рекуррентных формул следующего вида:

$$x_j^{(\mu)} = v_j^{(\mu)} \overline{x}_\mu + z_j^{(\mu)} \overline{x}_{\mu+1} + w_j^{(\mu)}; \; j = 0,..., m; \; \mu = 0,..., p-1. \qquad (3.20)$$

Здесь $\left\{v_j^{(\mu)}\right\}, \left\{z_j^{(\mu)}\right\}, \left\{w_j^{(\mu)}\right\}$ – решения следующих систем линейных уравнений с трехдиагональной матрицей:

$$\begin{cases} -a_{j+\mu m} v_{j-1}^{(\mu)} + b_{j+\mu m} v_j^{(\mu)} - c_{j+\mu m} v_{j+1}^{(\mu)} = 0; j = 1,..., m-1; \\ v_0^{(\mu)} = 1, v_m^{(\mu)} = 0. \end{cases} \qquad (3.21)$$

$$\begin{cases} -a_{j+\mu m} z_{j-1}^{(\mu)} + b_{j+\mu m} z_j^{(\mu)} - c_{j+\mu m} z_{j+1}^{(\mu)} = 0; j = 1,..., m-1; \\ z_0^{(\mu)} = 0, z_m^{(\mu)} = 1. \end{cases} \qquad (3.22)$$

$$\begin{cases} -a_{j+\mu m} w_{j-1}^{(\mu)} + b_{j+\mu m} w_j^{(\mu)} - c_{j+\mu m} w_{j+1}^{(\mu)} = f_{j+\mu m}; j = 1,..., m-1; \\ w_0^{(\mu)} = 0, w_m^{(\mu)} = 0. \end{cases} \qquad (3.23)$$

Системы (3.21) – (3.23) легко получаются из подстановки (3.20) в (3.13), решаются независимо для каждой μ-й подобласти обычным методом прогонки и допускают сокращение количества арифметиче-

ских операций за счет использования одной и той же трехдиагональной матрицы коэффициентов в (3.21) – (3.23).

После вычисления необходимых коэффициентов производится построение системы линейных уравнений для параметрических неизвестных $\{\overline{x}_\mu\}$. Для этого рассматриваются неиспользованные ранее уравнения системы (3.13) при $i = 0, m, 2m, ..., pm$, в которых с помощью формулы (3.20) исключаются все «непараметрические» неизвестные. В результате получим систему с трехдиагональной матрицей ($a_0 = 0, c_n = 0$):

$$\begin{aligned} -a_{\mu m} v_{m-1}^{(\mu)} \overline{x}_{\mu-1} + \left(b_{\mu m} - a_{\mu m} z_{m-1}^{(\mu)} - c_{\mu m} v_1^{(\mu)}\right) \overline{x}_\mu - c_{\mu m} z_1^{(\mu)} \overline{x}_{\mu+1} = \\ = f_{\mu m} + a_{\mu m} w_{m-1}^{(\mu-1)} + c_{\mu m} w_1^{(\mu)}; \mu = 0, ..., p. \end{aligned} \tag{3.24}$$

Решая эту систему обычным методом прогонки, найдем $\{\overline{x}_\mu\}$. Далее необходимо лишь воспользоваться формулой (3.20), чтобы определить остальные неизвестные задачи.

Хотя данный вычислительный алгоритм параллельного решения системы линейных алгебраических уравнений с трехдиагональной матрицей обладает высокой степенью параллелизма (вычисления по формулам (3.20) – (3.23) производятся с наивысшей для данной задачи степенью параллелизма p), тем не менее решение системы для параметрических неизвестных (3.24) выполняется последовательно только одним процессором. Кроме того, при использовании многопроцессорных вычислительных систем с распределенной памятью вычисление коэффициентов (3.24) и значений неизвестных по (3.20) также будет связано с дополнительными коммуникационными затратами, обусловленными необходимой передачей данных из локальной памяти других процессоров.

Оценим временные затраты на получение решения по данному алгоритму. На первом этапе (решение систем (3.21) – (3.23)) каждому процессору требуется около $13m$ арифметических операций, которые выполняются одновременно и независимо от других процессоров. Далее решается система (3.24) одним из процессоров. Для расчета коэффициентов системы (3.24) и нахождения ее решения требуется около $18p$ арифметических операций. Однако, если расчет ведется на МВС с распределенной памятью, то потребуется доставить в

локальную память этого процессора $6p$ чисел и возвратить остальным процессорам полученное решение системы (3.24) ($2p$ числа). На завершающем этапе каждый процессор одновременно и независимо ведет расчет по формуле (3.20), для чего понадобится $4m$ арифметических операций. В итоге получим оценку

$$T_p \approx (17m + 18p)t_a + 8pt_{comm}$$

и

$$S_p = \frac{T_1}{T_p} \approx \frac{8p/17}{1 + 18p^2/17n + 8\kappa appa * p^2/17n}.$$

Здесь $t_a = \max(t_{add}, t_{mult}, t_{div})$ – характерное время выполнения одной арифметической операции с плавающей точкой; t_{comm} – время на передачу одного числа между процессорными элементами, $\kappa appa = t_{comm} / t_a$.

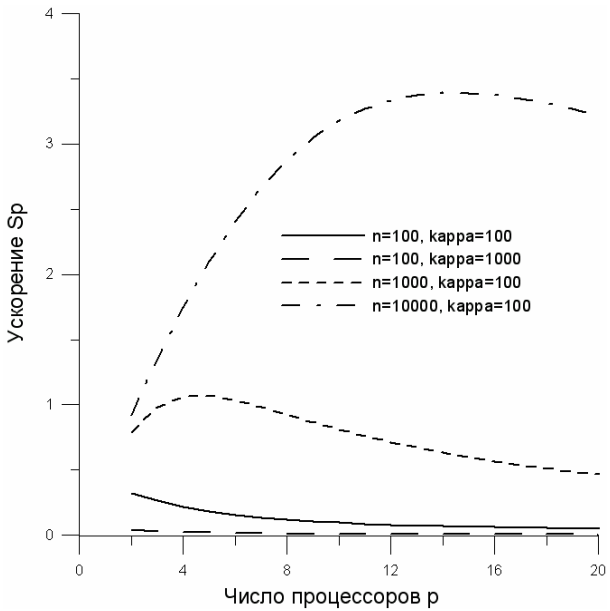


Рис. 3.14. Оценка ускорения метода параметрической прогонки

На рис. 3.14 приведены результаты расчетов ускорения по приведенной выше формуле. Из рисунка видно, что при небольших размерах задачи и низких показателях пропускной способности коммуникационной среды, связывающей узлы многопроцессорной вычислительной системы с распределенной памятью, использование рассмотренного метода параллельного решения системы (3.13) не является эффективным.

Более высоких показателей ускорения в этом методе можно добиться, лишь решая трехдиагональные системы большего размера на МВС с хорошей пропускной способностью коммуникационной среды.

Метод циклической редукции

При решении системы линейных алгебраических уравнений с трехдиагональной матрицей методом циклической редукции требуется, чтобы $n = 2^q, q \in \mathbb{N}$. Причём если первое и последнее уравнения в (3.11) имеют вид

$$b_0 x_0 - c_0 x_1 = f_0;$$
$$-a_n x_{n-1} + b_n x_n = f_n$$

и $c_0 \neq 0$ и $a_n \neq 0$, то для удобства реализации метода циклической редукции рекомендуется расширить систему путем добавления дополнительных уравнений $x_0 = 0$, $x_n = 0$ со сдвигом индексации соответствующих неизвестных.

Итак, пусть система трёхдиагональных уравнений (3.13) имеет вид

$$\begin{cases} x_0 = f_0, \\ -a_i x_{i-1} + b_i x_i - c_i x_{i+1} = f_i, i = 1,...,n-1; \quad n = 2^q; \\ x_n = f_n. \end{cases} \qquad (3.25)$$

Идея метода циклической редукции для решения этой задачи заключается в следующем. Циклически, на каждом шаге $k = 1,...,q-1$ производится с помощью линейных преобразований исключение неизвестных из системы (3.25) таким образом, что остаются неизвестные с индексами, кратными 2^k. Например, на первом шаге ($k = 1$) из

системы исключаются неизвестные с нечетными индексами следующим образом ($i = 2,4,...,n-2$):

$$\begin{aligned} &-a_{i-1}x_{i-2} + b_{i-1}x_{i-1} - c_{i-1}x_i = f_{i-1}, \\ &-a_i x_{i-1} + b_i x_i - c_i x_{i+1} = f_i, \\ &-a_{i+1}x_i + b_{i+1}x_{i+1} - c_{i+1}x_{i+2} = f_{i+1}. \end{aligned} \qquad (3.26)$$

Умножим первое уравнение (3.26) на $\frac{a_i}{b_{i-1}}$, а третье – на $\frac{c_i}{b_{i+1}}$ и прибавим их ко второму уравнению. В результате получим следующую систему $n/2+1$ уравнений, содержащую неизвестные с четными индексами:

$$\begin{cases} x_0 = f_0, \\ -a_i^{(1)}x_{i-2} + b_i^{(1)}x_i - c_i^{(1)}x_{i+2} = f_i^{(1)}, i = 2,4,...,n-2, \\ x_n = f_n. \end{cases} \qquad (3.27)$$

Здесь $a_i^{(1)} = \frac{a_{i-1}a_i}{b_{i-1}}, b_i^{(1)} = b_i - \frac{c_{i-1}a_i}{b_{i-1}} - \frac{a_{i+1}c_i}{b_{i+1}}, c_i^{(1)} = \frac{c_{i+1}c_i}{b_{i+1}},$

$$f_i^{(1)} = f_i + \frac{f_{i-1}a_i}{b_{i-1}} + \frac{f_{i+1}c_i}{b_{i+1}};\ i = 2,4,...,n-2.$$

Продолжая редукцию неизвестных, на последнем $(q-1)$-м шаге получим систему, содержащую только три уравнения:

$$\begin{cases} x_0 = f_0, \\ -a_{n/2}^{(q-1)}x_0 + b_{n/2}^{(q-1)}x_{n/2} - c_{n/2}^{(q-1)}x_n = f_{n/2}^{(q-1)}, \\ x_n = f_n. \end{cases} \qquad (3.28)$$

Найденное из (3.28) значение

$$x_{n/2} = \frac{f_{n/2}^{(q-1)} + a_{n/2}^{(q-1)}x_0 + c_{n/2}^{(q-1)}x_n}{b_{n/2}^{(q-1)}}$$

может затем использоваться для определения значений неизвестных $x_{n/4}, x_{3n/4}$ в редуцированной на $(q-2)$-м шаге системе

$$\begin{cases} x_0 = f_0, \\ -a_{n/4}^{(q-2)} x_0 + b_{n/4}^{(q-2)} x_{n/4} - c_{n/4}^{(q-2)} x_{n/2} = f_{n/4}^{(q-2)}, \\ -a_{n/2}^{(q-2)} x_{n/4} + b_{n/2}^{(q-2)} x_{n/2} - c_{n/2}^{(q-2)} x_{3n/4} = f_{n/2}^{(q-2)}, \\ -a_{3n/4}^{(q-2)} x_{n/2} + b_{3n/4}^{(q-2)} x_{3n/4} - c_{3n/4}^{(q-2)} x_n = f_{3n/4}^{(q-2)}, \\ x_n = f_n. \end{cases} \qquad (3.29)$$

Пользуясь таким приемом, можно последовательно найти остальные неизвестные с индексами, кратными $n/8, n/16, ..., 2, 1$.

Следовательно, процедура циклической редукции включает вычисления для шагов $k = 1, ..., q-1$ по рекуррентным формулам новых коэффициентов и правых частей:

$$a_i^{(k)} = P_i a_{i-2^{k-1}}^{(k-1)}; \quad b_i^{(k)} = b_i^{(k-1)} - P_i c_{i-2^{k-1}}^{(k-1)} - Q_i a_{i+2^{k-1}}^{(k-1)}; \quad c_i^{(k)} = Q_i c_{i+2^{k-1}}^{(k-1)};$$

$$f_i^{(k)} = f_i^{(k-1)} + P_i f_{i-2^{k-1}}^{(k-1)} + Q_i f_{i+2^{k-1}}^{(k-1)}; \quad P_i = \frac{a_i^{(k-1)}}{b_{i-2^{k-1}}^{(k-1)}}; Q_i = \frac{c_i^{(k-1)}}{b_{i+2^{k-1}}^{(k-1)}}; \qquad (3.30)$$

для $i = 2^k, 2 \times 2^k, ..., n - 2^k$ при $a_i^{(0)} = a_i, b_i^{(0)} = b_i, c_i^{(0)} = c_i, f_i^{(0)} = f_i$

с последующим замещением решения для $k = q, q-1, ..., 2, 1$ из

$$x_i = \frac{f_i^{(k-1)} + a_i^{(k-1)} x_{i-2^{k-1}} + c_i^{(k-1)} x_{i+2^{k-1}}}{b_i^{(k-1)}}, \qquad (3.31)$$

где $i = 2^{k-1}, 2 \times 2^{k-1}, ..., n - 2^{k-1}$.

На рис. 3.15 представлена диаграмма маршрутизации прямого и обратного хода циклической редукции для $n = 8$. Оценим временные затраты данного алгоритма.

Для расчета всех коэффициентов каждого уравнения редуцированной системы потребуется 12 арифметических операций, всего таких расчетов необходимо сделать $(n/2 - 1) + (n/4 - 1) + ... + 1 = n - q - 1$. При вычислении значения одного неизвестного используется 5 арифметических операций, а всего их будет $\approx 5n$. В итоге метод циклической редукции, также как и

метод прогонки, является экономичным методом*, которому требуется около $17n$ арифметических операций.

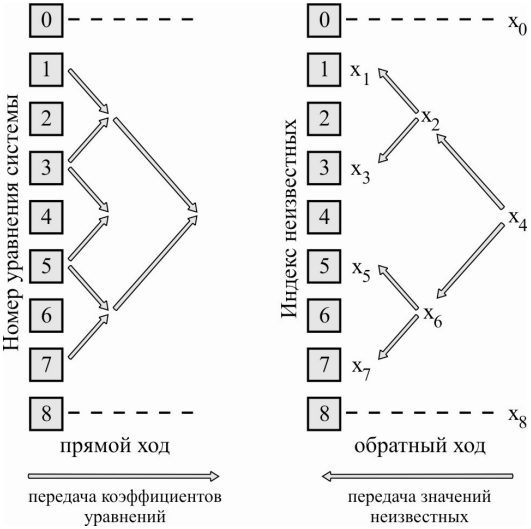


Рис. 3.15. Диаграмма маршрутизации прямого и обратного хода метода циклической редукции

Этот алгоритм обладает хорошим параллелизмом (рис. 3.16). Во-первых, операции исключения (3.30) независимы и могут выполняться параллельно. Во-вторых, расчет значений неизвестных по формулам (3.31) проводится одновременно. Однако если используется многопроцессорная система с распределенной памятью и обычная схема декомпозиции данных по процессорам, то на этапе редукции понадобится передача полученных коэффициентов в локальную память других процессорных элементов. Кроме того, на завершающих шагах прямого хода будет удваиваться число простаивающих процессоров. Похожая картина наблюдается и при расчете значений неизвестных. Сначала вычисляется значение $x_{n/2}$ на одном процессоре, затем пересылается остальным, далее одновременно на разных процессорных элементах рассчитываются $x_{n/4}, x_{3n/4}$ и т.д.

* Численный метод называется экономичным, если он корректен и число арифметических операций в нем пропорционально размеру задачи.

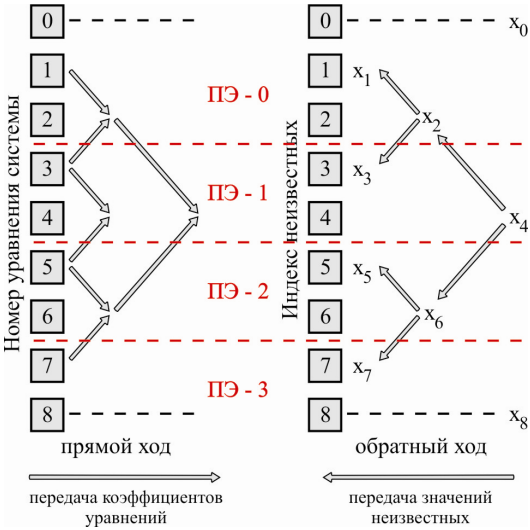


Рис. 3.16. Параллельная реализация метода циклической редукции

Оценим трудоёмкость полученной параллельной реализации алгоритма метода циклической редукции для случая $p = 2^r << n$. Пусть при инициализации в локальной памяти каждого процессорного элемента размещены по $m = n / p$ уравнений системы (3.25) (за исключением первого или последнего, где на одно уравнение больше). Тогда на этапе редукции (прямом ходе) временные затраты на вычисления коэффициентов оцениваются как $12 \times m \times t_a$, к этому нужно добавить время на выполняющиеся одновременно на каждом шаге редукции пересылки набора коэффициентов $F_{i+2^{k-1}} = \left\{ a_{i+2^{k-1}}, b_{i+2^{k-1}}, c_{i+2^{k-1}}, f_{i+2^{k-1}} \right\}$ между процессорными элементами, номера которых отличаются на единицу. С учетом оценки времени выполнения параллельного алгоритма на этапе вычисления неизвестных получим

$$T_p \approx 17m \times t_a + 5q \times t_{comm}$$

и ускорение

$$S_p = \frac{T_1}{T_p} \approx \frac{p}{1 + \frac{5\kappa appa * p \log_2(n)}{17n}}, \quad \kappa appa = \frac{t_{comm}}{t_a}.$$

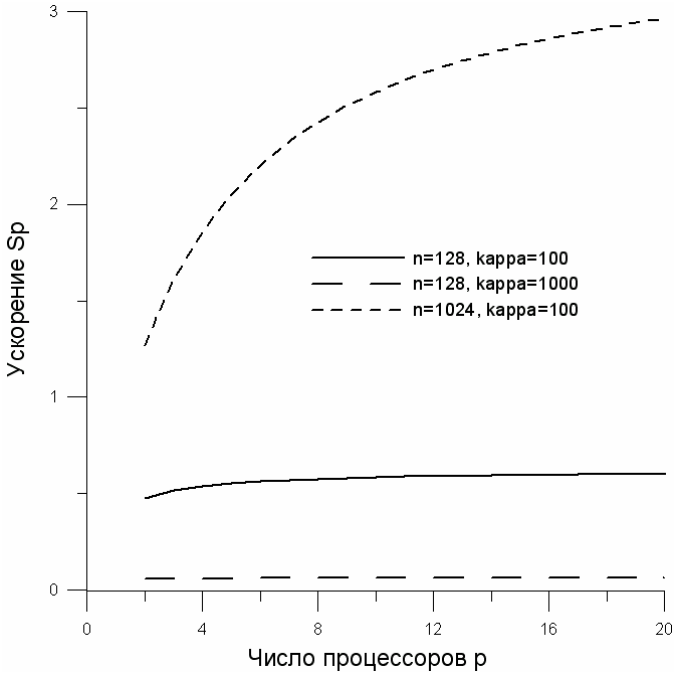


Рис. 3.17. Оценка ускорения метода циклической редукции

На рис. 3.17 представлены графики оценки ускорения параллельного алгоритма циклической редукции. Сравнивая рис. 3.14 и 3.17, можно сделать вывод, что параллельный алгоритм метода циклической редукции при небольших размерах задачи ($n \sim 10^2$) и для многопроцессорных систем с медленной скоростью передачи данных между вычислительными узлами ($\kappa appa = t_{comm} / t_a \sim 10^3$) так же, как и алгоритм параметрической прогонки, не является эффективным. При увеличении размера задачи до $n \sim 10^3$ рассмотренный параллельный вариант метода циклической редукции предпочтительнее,

поскольку позволяет получить более высокие оценки величины ускорения параллельной программы.

Рассмотрим вариант метода циклической редукции, поддерживающий параллелизм операций на максимально высоком уровне, что обеспечивается за счет другого способа организации вычислений на фазе редукции. На первом шаге из уравнений системы поочередно исключаются неизвестные с четными и нечетными номерами. В результате получаются две независимые подсистемы. На втором шаге редуцируется похожим образом каждая полученная подсистема, что приводит в результате к четырем независимым подсистемам, содержащим по $(n-1)/4$ неизвестных. На последнем шаге этапа система уравнений (3.25) преобразуется к системе с диагональной матрицей (или к $n+1$ независимым уравнениям для отдельных неизвестных), решение которой можно получить, проводя одновременно вычисления по явным формулам.

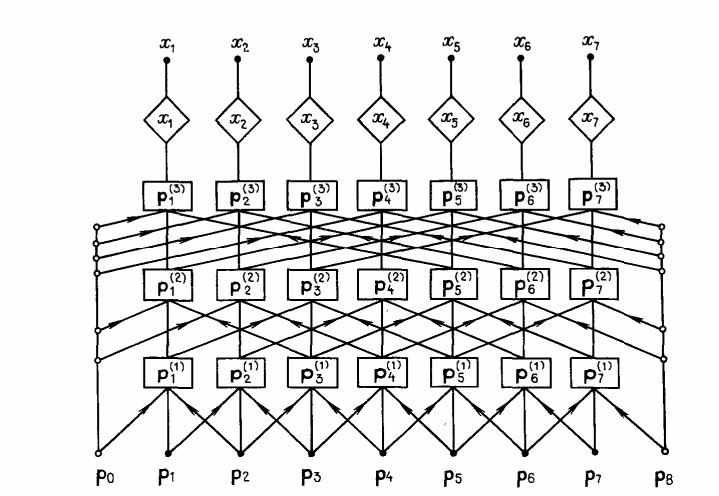


Рис. 3.18. Диаграмма маршрутизации метода циклической редукции без обратного хода [1]

На рис. 3.18 приведена диаграмма маршрутизации построенного алгоритма, который часто называют методом циклической редукции без обратного хода. Из диаграммы видно, что рассмотренный метод обладает максимальным параллелизмом, что стало возможным за счет увеличения общего числа арифметических операций. Следует

также отметить, что одновременно существенно возросли и коммуникационные затраты на этапе редукции, связанные с более активной пересылкой коэффициентов и правых частей линейных уравнений.

Таким образом, рассматривая параллельные варианты методов прогонки и циклической редукции, следует отметить, что эти методы не следует использовать в качестве алгоритмов, требующих распараллеливания процедуры решения небольших ($n \sim 10^2$) систем с трехдиагональными матрицами. В настоящее время в параллельных вычислениях произошел переход от мелкозернистых и с большими коммуникационными затратами параллельных алгоритмов к алгоритмам, которые имеют крупноблочный параллелизм с очень малым числом обменов за счет применения в параллельных ветвях алгоритма одновременно выполняющихся последовательных шагов.

Для блочных трехдиагональных систем, размер которых на два-три порядка выше числа используемых процессоров, целесообразно рассматривать технологию решения, в которой трехдиагональные подсистемы решаются одновременно (например обычным методом прогонки), а не технологию, в которой каждая отдельная трехдиагональная подсистема решалась бы параллельно на всех процессорных элементах.

Параллельная реализация метода циклической редукции, рассмотренная выше, легко обобщается и на случай систем с пятидиагональными матрицами.

4. ИТЕРАЦИОННЫЕ МЕТОДЫ РЕШЕНИЯ ЛИНЕЙНЫХ СИСТЕМ

Итерационные методы для решения линейных систем $A\vec{x} = \vec{b}$ с невырожденной квадратной матрицей $A \in \mathbb{R}^{n \times n}$ используя заданное начальное приближение $\vec{x}_0 \in \mathbb{R}^n$ выполняют его последовательное улучшение до тех пор, пока приближенное решение не будет найдено с требуемой точностью. Итерации заканчиваются, когда норма невязки

$$\left\| \vec{b} - A\vec{x}_k \right\| = \max_{1 \le i \le n} \left| b_i - \sum_{j=1}^{n} a_{ij} x_j^{(k)} \right|$$

или другая мера ошибки (например, относительная или абсолютная погрешность определения компонент решения) не станет малой.

Итерационные методы обычно используются для решения систем уравнений, количество которых велико. Эти методы практически незаменимы при решении больших разреженных и плохо обусловленных систем*, поскольку строятся таким образом, чтобы погрешность метода во время поиска решения не накапливалась. Последовательные приближения к решению в таких методах обычно генерируются выполнением умножений матрицы на вектор. Итерационные методы не гарантируют получения решения для любой системы уравнений. Однако когда они дают решение, то оно обычно получается с меньшими затратами, чем с использованием прямых методов.

Итерационные методы для решения линейных систем включают следующие базовые операции линейной алгебры:

- линейную комбинацию векторов (**saxpy**);
- скалярное произведение векторов (**dot**);
- умножение матрицы на вектор (**gaxpy**);
- решение систем с треугольными матрицами.

При параллельной реализации таких операций (главы 2 и 3) производится декомпозиция данных и операций по числу используемых

---

* Плохо обусловленной называется система линейных уравнений, имеющая единственное решение, в которой малые изменения в коэффициентах матрицы приводят к значительному изменению решения. Это имеет место в случае, когда число обусловленности матрицы системы $cond(A) = \|A\| \times \|A^{-1}\| \gg 1$.

параллельных процессов, которая сопровождается передачей данных между процессорными элементами для обеспечения локальных вычислений.

4.1. Метод Якоби

Задавая начальное приближение $\vec{x}_0$, в методе Якоби новое $(k+1)$-е приближение к точному решению для каждой компоненты рассчитывается по следующей формуле ($a_{ii} \neq 0, i = 1,2,...,n$):

$$x_i^{(k+1)} = \frac{b_i - \sum\limits_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)}}{a_{ii}}, i = 1,...,n; k = 0,1,... . \qquad (4.1)$$

Здесь i – номер компоненты вектора приближенного решения $\vec{x}_k$; k – номер итерации.

Пусть D – диагональная матрица, образованная диагональными элементами A, а L и U – нижняя и верхняя треугольные матрицы вида

$$L = \begin{pmatrix} 0 & 0 & \dots & 0 \\ a_{21} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}, \; U = \begin{pmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}, \; D + L + U = A,$$

тогда (4.1) можно записать в виде

$$\vec{x}_{k+1} = -D^{-1}\left(L + U\right)\vec{x}_k + D^{-1}\vec{b} . \qquad (4.2)$$

Для корректности метода Якоби требуются ненулевые диагональные элементы матрицы A, что можно обеспечить перестановкой строк или столбцов такой матрицы, если это необходимо. При реализации метода Якоби на компьютере в памяти должны храниться все компоненты векторов $\vec{x}_k$ и $\vec{x}_{k+1}$, поскольку ни один из компонентов $\vec{x}_k$ не может быть переписан до тех пор, пока новое приближение $\vec{x}_{k+1}$ не будет получено. Кроме того, следует обратить внимание, что

по формулам (4.1) или (4.2) все компоненты следующего приближения $\vec{x}_{k+1}$ могут быть рассчитаны одновременно и независимо, что открывает большие возможности при построении параллельной версии такого алгоритма.

Конечно, метод Якоби сходится не всегда, но если матрица A имеет строгое диагональное преобладание

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|, i = 1,...,n \text{ или } |a_{jj}| > \sum_{\substack{i=1 \\ i\neq j}}^{n} |a_{ij}|, j = 1,...,n,$$

то этого достаточно для сходимости метода, хотя сходимость может быть очень медленной. Кроме того, для проверки сходимости метода Якоби используются необходимые и достаточные условия: корни характеристического уравнения

$$\det \begin{pmatrix} \lambda a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \lambda a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & \lambda a_{nn} \end{pmatrix} = 0$$

должны по модулю быть меньше единицы.

В качестве критерия завершения итерационного процесса рассматривают следующее условие:

$$\left\| \vec{x}_{k+1} - \vec{x}_k \right\| = \max_i \left| x_i^{(k+1)} - x_i^{(k)} \right| < \varepsilon \ \underline{\textbf{и}} \ \left\| Ax_{k+1} - b \right\| < \varepsilon, \qquad (4.3)$$

где $\vec{x}_k$ – приближенное значение решения на k-й итерации численного метода.

Пользуясь формулой (4.1), оценим временные затраты на выполнение одной итерации рассмотренного алгоритма метода Якоби (4.1):

$$T_1 \approx n \cdot \left((n-1) \cdot \left(t_{add} + t_{mult} \right) + t_{add} + t_{div} \right) \approx 2 \cdot n^2 \cdot t_a; (n >> 1),$$

где t_a – обобщенное время одной арифметической операции (+, –, *, /).

При построении параллельной версии итерационного метода Якоби будем пользоваться подходом, опирающимся на декомпозицию данных, т.е. для p-процессорной параллельной системы будем на-

значать на расчет *(k+1)*-го приближения по формуле (4.1) по *n/p* компонентов вектора $\vec{x}_{k+1}$, для чего понадобится *n/p* строк матрицы *A* и *n/p* компонентов вектора $\vec{b}$. К такому же способу организации параллельных вычислений можно прийти, если пользоваться стандартной технологией разработки параллельных программ, подробно описанной во введении.

Действительно, на этапе декомпозиции в качестве фундаментальной мелкозернистой подзадачи выберем задачу расчета одного компонента вектора $\vec{x}_{k+1}$ по известным значениям вектора предыдущего приближения $\vec{x}_k$, коэффициентов матрицы A и вектора правой части $\vec{b}$ (4.1).

При проектировании коммуникаций между подзадачами становится очевидным, что они не требуются, поскольку расчеты каждого компонента $\vec{x}_{k+1}$ на одной итерации ведутся независимо и одновременно. Однако следует отметить, что при выполнении любого итерационного метода необходимо контролировать выполнение условия (4.3), а для этого (для вычисления нормы ошибки и нормы невязки) нужно располагать всеми компонентами полученного нового приближения $\vec{x}_{k+1}$, а также значениями $\vec{x}_k$, A, $\vec{b}$. Таким образом, потребуется сборка $\vec{x}_{k+1}$ либо на одном или на всех процессорных элементах с последующим расчетом условия (4.3) и принятия решения о продолжении итерационного процесса.

Первый способ (сборка $\vec{x}_{k+1}$ на одном процессорном элементе) использует меньшее количество межпроцессорных коммуникаций, однако, когда будут рассчитываться значения норм в (4.3), остальные процессоры будут простаивать, что снизит эффективность параллельной программы. Кроме того, после вычисления логического значения по неравенствам (4.3) процессорный элемент, который проводил вычисления, должен передать сообщение остальным процессорным элементам системы о принятом решении о продолжении или остановке итерационного процесса, на что также потребуются обменные операции.

При сборке $\vec{x}_{k+1}$ на каждом процессорном элементе потребуется несколько большее количество межпроцессорных коммуникаций, но многие из них при топологии многопроцессорной вычислительной

системы с высокой степенью связности могут проходить одновременно (здесь также требуется, чтобы на каждом ПЭ были $\vec{x}_k$, A, $\vec{b}$). Дублирование вычисления значений норм и проверка условия (4.3) на каждом процессорном элементе так же, как и в предыдущем случае, снизят эффективность параллельной программы, однако решение о продолжении или остановке итерационного процесса выполняется синхронно без организации дополнительной передачи данных. Кроме того, поскольку для продолжения вычислений на следующей итерации требуется иметь текущее приближение, то этот способ имеет преимущество.

Таким образом, хотя выбранный способ декомпозиции обеспечивает получение нового приближения к точному решению системы $A\vec{x} = \vec{b}$ без межпроцессорных коммуникаций, проверка условия завершения итерационного процесса (4.3) является узким местом разрабатываемого параллельного алгоритма/программы. Это связано с наличием глобальных коммуникаций в массиве фундаментальных подзадач, которые необходимы для расчета условия (4.3) и для обновления вектора последнего приближения $\vec{x}_k$ в (4.1). Также заметим, что требование хранения всей матрицы A на одном или на каждом ПЭ ограничивает размер задачи n, который может быть значительным.

Рассмотрим схему укрупнения мелкозернистых фундаментальных подзадач, при которой каждый укрупненный блок содержит по *n/p* таких подзадач. Для такой схемы укрупнения расчет *n/p* компонентов $\vec{x}_{k+1}$ в каждом блоке не требует хранения всей матрицы A, а лишь *n/p* ее строк (см. рис. 4.1). Это же касается вектора $\vec{b}$ – для расчетов нужны лишь *n/p* его компонентов. Вектор x_k должен быть представлен всеми своими компонентами (рис. 4.1).

Параллельная версия итерационного метода Якоби в таком случае реализуется следующим образом:

- *Этап инициализации*: на каждом процессорном элементе ПЭμ размещается A_μ, $\left(\vec{b}\right)_\mu$ ($\mu = 1,..., p$) и $\vec{x}_0$, где A_μ, $\left(\vec{b}\right)_\mu$ содержат строки матрицы A и компоненты вектора $\vec{b}$ с индексами $i = 1 + (\mu - 1)\frac{n}{p},..., \mu\frac{n}{p}; \mu = 1,..., p$.

- *Этап параллельных вычислений*:

для $k = 0,1,2,...$ **выполнять на каждом** ПЭμ**:**

{1} расчет $\left(\vec{x}_{k+1}\right)_{\mu}$ **по формуле** (4.1)**;**

{2} расчет $\left\|\left(\vec{x}_{k+1}\right)_{\mu} - \left(\vec{x}_{k}\right)_{\mu}\right\|$ **и** $\left\|\left(\vec{b}\right)_{\mu} - A_{\mu}\vec{x}_{k}\right\|$**;**

{3} пересылка полученных значений норм вместе с векторами $\left(\vec{x}_{k+1}\right)_{\mu}$ **на остальные ПЭ;**

{4} обновление вектора последнего приближения и проверка каждым ПЭ выполнения условия (4.3). **В случае, если условие выполнено, то выход из цикла.**

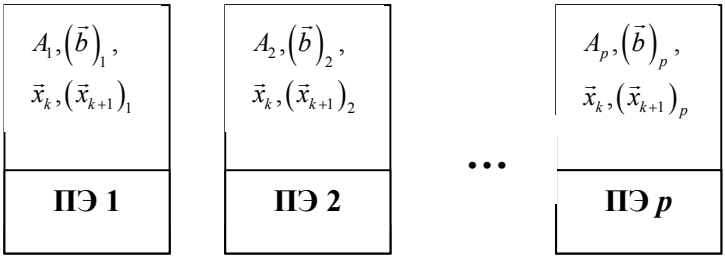


Рис. 4.1. Распределение исходных данных ($k = 0$) по процессорным элементам

Оценим временные затраты разработанного параллельного алгоритма на одной итерации без учета этапа инициализации $\left(n / p \gg 1\right)$:

$$T_p \approx \underset{\{1\}}{2\frac{n^2}{p}t_a} + \underset{\{2\}}{4\frac{n}{p}t_a} + \underset{\{3\}}{\left(p-1\right)\left(\frac{n}{p}+2\right)t_{comm}} \approx 2\frac{n^2}{p}t_a + \left(p-1\right)\frac{n}{p}t_{comm}.$$

При оценке коммуникационных затрат предполагается, что каждый ПЭ одновременно рассылает по $\left(n / p + 2\right)$ числа остальным процессорным элементам.

Тогда получим оценку ускорения и эффективности алгоритма:

$$S_p = \frac{T_1}{T_p} \approx \frac{2n^2 t_a}{2\frac{n^2}{p} t_a + (p-1)\frac{n}{p} t_{comm}} = \frac{p}{1 + \frac{(p-1)\kappa}{2n}}; \qquad (4.4)$$

$$E_p = \frac{S_p}{p} \approx \frac{1}{1 + \frac{(p-1)\kappa}{2n}}, \kappa = \frac{t_{comm}}{t_a}.$$

Из формул (4.4) следует, что можно достичь высокой эффективности распараллеливания, если $n / p \gg \kappa \gg 1$, $\kappa = t_{comm} / t_a$ – показатель, характеризующий соотношение быстродействия передачи данных по межпроцессорной сети к быстродействию процессоров. Следует отметить еще одно свойство итерационного метода Якоби: последовательная и параллельная версии алгоритма выполняют одинаковый объем вычислительной работы, что характеризуется одинаковым количеством итераций для получения приближенного решения с заданной точностью.

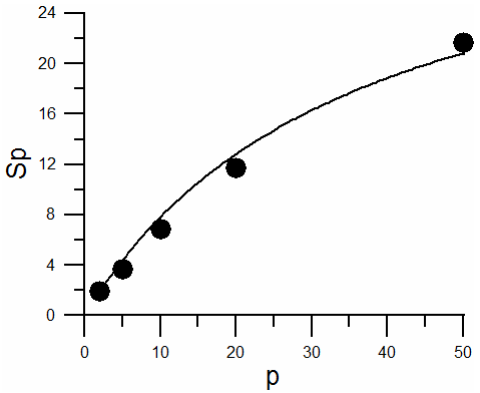


Рис. 4.2. Ускорение параллельного метода Якоби при решении СЛАУ с полнозаполненной матрицей (n = 5000). Линия – оценка по формуле (4.4) при к = 140. Значки – время расчета по параллельной программе на кластере СКИФ Cyberia

На рис. 4.2 представлены результаты зависимости оценки ускорения описанного выше параллельного алгоритма и построенной по нему параллельной программы для n = 5000 и $\varepsilon = 5 \cdot 10^{-6}$ от числа используемых процессоров p. Численные расчеты проведены на супер-

компьютере СКИФ Cyberia при решении системы линейных уравнений с матрицей, все элементы которой равны единице за исключением диагональных, имеющих значение 1.1*(n – 1).

Для полученных по формуле (4.4) теоретических оценок ускорения параллельного алгоритма к = 140. Из рисунка видно, что рост ускорения параллельной программы с увеличением числа используемых процессорных элементов p уменьшается, что связано (см. левую часть (4.4)) с сокращением объема вычислений при неизменном объеме пересылаемых между процессорными элементами данных.

4.2. Метод Зейделя и метод верхней релаксации

Более быструю сходимость при последовательных вычислениях имеет метод Зейделя. Оценки показывают, что скорость сходимости метода Зейделя в случае диагонального преобладания матрицы A в среднем в 2–3 раза выше, чем метода Якоби.

Покомпонентная форма записи метода Зейделя имеет вид ($a_{ii} \neq 0, i = 1,2,...,n$):

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}}, i = 1,2,...,n. \qquad (4.5)$$

Однако это не единственная форма записи этого метода, основная идея которого заключается в использовании только что вычисленных на $(k+1)$-й итерации компонент вектора $\vec{x}_{k+1}$ для расчета остальных. Например, можно рассмотреть и такую форму:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k+1)}}{a_{ii}}, i = n, n-1,...,1. \qquad (4.6)$$

Также можно рассматривать и другой порядок изменения значения индекса i, главное, чтобы только что вычисленные $x_i^{(k+1)}$ сразу же использовались в вычислительном процессе.

Матрично-векторная форма записи (4.5) и (4.6) имеет вид:

$$\vec{x}_{k+1} = (D+L)^{-1}\left(\vec{b} - U\vec{x}_k\right), \tag{4.7}$$

$$\vec{x}_{k+1} = (D+U)^{-1}\left(\vec{b} - L\vec{x}_k\right), \tag{4.8}$$

где D, L, U – диагональная, нижнетреугольная и верхнетреугольная матрицы, составленные из элементов матрицы A, причем $D + L + U = A$.

Метод Зейделя сходится при любом начальном приближении и любой правой части $\vec{b}$, если матрица A имеет строгое диагональное преобладание. Необходимые и достаточные условия сходимости метода Зейделя заключаются в требовании, чтобы все корни характеристических уравнений

$$\det\begin{pmatrix} \lambda a_{11} & a_{12} & \dots & a_{1n} \\ \lambda a_{21} & \lambda a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda a_{n1} & \lambda a_{n2} & \dots & \lambda a_{nn} \end{pmatrix} = 0 \text{ для метода (4.7)}$$

или

$$\det\begin{pmatrix} \lambda a_{11} & \lambda a_{12} & \dots & \lambda a_{1n} \\ a_{21} & \lambda a_{22} & \dots & \lambda a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & \lambda a_{nn} \end{pmatrix} = 0 \text{ для метода (4.8)}$$

были бы по модулю меньше единицы.

Анализируя формулы (4.5) и (4.6), можно заметить, что в отличие от метода Якоби для хранения $\vec{x}_{k+1}$ и $\vec{x}_k$ требуется один вектор, и расчет по формулам метода Зейделя подразумевает лишь последовательное выполнение входящих в формулы арифметических операций. Дело в том, что формулы (4.5) и (4.6) являются рекуррентными, т.е. для расчета $x_i^{(k+1)}$ в (4.5) нужно предварительно на $(k+1)$-й итерации рассчитать значения $x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_{i-1}^{(k+1)}$.

Так, если использовать представленную на рис. 4.1 схему распределения исходных данных по процессорным элементам, то каждое

только что рассчитанное значение $x_i^{(k+1)}$ в соответствии с (4.5) немедленно должно быть отправлено на остальные процессорные элементы, которые вынуждены простаивать в ожидании этого значения, и, следовательно, расчет каждого $x_i^{(k+1)}$ ведется одним ПЭ при простаивающих остальных (последовательное выполнение алгоритма метода Зейделя (4.5)).

Для преодоления этой проблемы в практике параллельных вычислений часто применяют так называемую асинхронную, или хаотическую релаксацию, когда при получении компонент вектора приближенного решения на следующей итерации используют последние значения $\vec{x}_k$ и $\vec{x}_{k+1}$, имеющиеся на каждом процессорном элементе (рис. 4.1), а передача вычисленных значений другим процессорным элементам осуществляется после завершения итерации. Этот подход может быть эффективным, однако порядок выполнения вычислительной работы в нем отличается от организации расчетов в оригинальном методе Зейделя, что существенно усложняет анализ сходимости полученного параллельного метода решения систем линейных алгебраических уравнений.

Заметим, что при решении СЛАУ с плотно заполненной (незначительное число нулевых элементов) матрицей, размер которой позволяет ее располагать в оперативной памяти процессорных элементов, можно воспользоваться матрично-векторной формой записи метода Зейделя: (4.7) или (4.8). Из этих формул видно, что расчет компонент вектора $\vec{x}_{k+1}$ может вестись при решении на каждой итерации системы с верхне- или нижнетреугольной матрицей. Выше было отмечено, что для таких систем параллельные методы оказываются лишь умеренно эффективными.

Рассмотрим важную модификацию метода Зейделя. Обозначим через $\tilde{x}_i^{(k+1)}$ правую часть (4.5), задающую итерационное приближение метода Зейделя, и определим

$$x_i^{(k+1)} = x_i^{(k)} + \omega\left(\tilde{x}_i^{(k+1)} - x_i^{(k)}\right), i = 1,2,...,n\,. \qquad (4.9)$$

Здесь $\omega > 0$ – параметр релаксации. Если подставить выражение для $\tilde{x}_i^{(k+1)}$ в (4.9), то после преобразования можно получить

$$a_{ii}x_i^{(k+1)} + \omega\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} = (1-\omega)x_i^{(k)} - \omega\sum_{j=i+1}^{n} a_{ij}x_j^{(k)} + \omega b, i = 1,2,...,n$$

или (если $a_{ii} \neq 0, i = 1,2,...,n$)

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right], i = 1,2,...,n. \quad (4.10)$$

В матрично-векторном виде можно это записать как

$$\vec{x}_{k+1} = \left(D + \omega L\right)^{-1}\left[(1-\omega)D - \omega U\right]\vec{x}_k + \omega\left(D + \omega L\right)^{-1}\vec{b}. \qquad (4.11)$$

Полученные формулы (4.10) и (4.11) определяют итерационный метод последовательной верхней релаксации SOR (*successive over relaxation*), который при $\omega = 1$ в (4.10) и (4.11) сводится к итерациям метода Зейделя. Заметим также, что матрица $\left(D + \omega L\right)$ – невырожденная в силу принятого предположения о диагональных элементах матрицы A.

Итерации метода SOR сходятся при любом начальном приближении, если A – симметричная положительно определенная матрица и $\omega \in (0,2)$. При использовании оптимального значения параметра релаксации скорость сходимости SOR на порядок выше, чем скорость сходимости метода Зейделя.

Параллельная реализация метода SOR для линейных систем с плотно заполненной матрицей осуществляется аналогично распараллеливанию метода Зейделя, т.е. на каждой итерации решаются параллельным методом, описанным в главе 3, треугольные системы вида

$$\left(D + \omega L\right)\vec{x}_{k+1} = \left[(1-\omega)D - \omega U\right]\vec{x}_k + \omega\vec{b}, \quad k = 0,1,2,... \qquad (4.12)$$

или используются асинхронные или хаотические релаксации для вычислений по формуле (4.10).

Применение асинхронного подхода в качестве способа распараллеливания метода Зейделя или метода SOR ведет, как правило, к увеличению общего объема числа итераций при заданной точности вычислений по сравнению с соответствующими последовательными алгоритмами, причем при увеличении степени декомпозиции исход-

ных данных (с ростом числа используемых процессорных элементов) количество итераций повышается в отличие от метода Якоби, для которого общее число итераций в последовательной и параллельной версиях алгоритма совпадают.

4.3. Итерационные методы вариационного типа

Итерационные методы, ставящие в соответствие задаче решения СЛАУ эквивалентную задачу минимизации некоторого функционала, называются итерационными методами вариационного типа. Среди них выделяют методы наискорейшего спуска и метод минимальных невязок. Рассмотрим применение указанных методов вариационного типа для решения систем $A\vec{x} = \vec{b}$ с плотно заполненной симметричной матрицей положительного типа. В случае несимметричной матрицы A при необходимости можно по Гауссу провести симметризацию исходной системы ($A^T(A\vec{x}) = A^T\vec{b}$, A^T – транспонированная матрица).

Метод минимальных невязок вводится следующим образом:

$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{r}_k, \quad k = 0,1,2,\ldots, \tag{4.13}$$

где $\vec{r}_k = \vec{b} - A\vec{x}_k$ – вектор невязки, начальное приближение $\vec{x}_0$ задано. Параметр α_k на каждой итерации подбирается так, чтобы минимизировать евклидову норму невязки $\|\vec{r}_{k+1}\| = \sqrt{(\vec{r}_{k+1}, \vec{r}_{k+1})}$.

Умножим (4.13) на матрицу A слева и после вычитания из правой и левой частей равенства вектора $\vec{b}$ получим

$$\vec{r}_{k+1} = \vec{r}_k - \alpha_k A\vec{r}_k \quad \text{и}$$

$$\|\vec{r}_{k+1}\|^2 = (\vec{r}_{k+1}, \vec{r}_{k+1}) = (\vec{r}_k, \vec{r}_k) - 2\alpha_k (A\vec{r}_k, \vec{r}_k) + \alpha_k^2 (A\vec{r}_k, A\vec{r}_k). \tag{4.14}$$

В последнем соотношении параметр α_k будет выбираться из условия минимума нормы невязки на следующей $(k+1)$-й итерации. Приравняв нулю производную по α_k от правой части (4.14), получим

$$\alpha_k = \frac{(A\vec{r}_k, \vec{r}_k)}{(A\vec{r}_k, A\vec{r}_k)}. \tag{4.15}$$

Градиентные методы или методы наискорейшего спуска ищут следующее приближение к точному решению $\vec{x}_{k+1}$ в итерационном процессе на основе предыдущего $\vec{x}_k$ в направлении градиента функционала $\varphi(\vec{x}) = \frac{1}{2}\left(\vec{x}, A\vec{x}\right) - \left(\vec{x}, \vec{b}\right)$ по правилу:

$$\vec{x}_{k+1} = \vec{x}_k - \alpha_k \vec{p}_k, \quad k = 0,1,2,..., \tag{4.16}$$

где векторы $\vec{p}_k$ определяют направление поиска минимума, а скаляры α_k – расстояние, на которое осуществляется продвижение к точному решению $\vec{x}^*$ по направлению $\vec{p}_k$.

Если в качестве вектора $\vec{p}_k$ выбрать $\varphi'(\vec{x}_k) = A\vec{x}_k - \vec{b} = -\vec{r}_k$, то значения $\{\alpha_k\}$ находятся из условия минимума функционала:

$$\varphi(\vec{x}_{k+1}) = \varphi(\vec{x}_k + \alpha_k \vec{r}_k), \text{ т.е. } \varphi(\vec{x}_k + \alpha_k \vec{r}_k) = \min_{\{\alpha\}}\left[\varphi(\vec{x}_k + \alpha \vec{r}_k)\right].$$

Для известных $\vec{x}_k$ и $\vec{r}_k$

$$\varphi(\vec{x}_k + \alpha\vec{r}_k) = f(\alpha) = \frac{1}{2}\left((\vec{x}_k + \alpha\vec{r}_k), A(\vec{x}_k + \alpha\vec{r}_k)\right) - \left(\vec{b}, (\vec{x}_k + \alpha\vec{r}_k)\right) =$$

$$= \frac{1}{2}\left(\vec{x}_k, A\vec{x}_k\right) + \alpha\left(\vec{r}_k, A\vec{x}_k\right) + \frac{1}{2}\alpha^2\left(\vec{r}_k, A\vec{r}_k\right) - \alpha\left(\vec{b}, \vec{r}_k\right) - \left(\vec{b}, \vec{x}_k\right) =$$

$$= \frac{1}{2}\alpha^2\left(\vec{r}_k, A\vec{r}_k\right) - \alpha\left(\vec{r}_k, \left(\vec{b} - A\vec{x}_k\right)\right) + \frac{1}{2}\left(\vec{x}_k, \left(A\vec{x}_k - 2\vec{b}\right)\right).$$

В силу положительной определенности $A\left(\vec{r}_k, A\vec{r}_k\right) > 0$ минимум $f(\alpha)$ будет достигаться при значении α, обращающем в 0 производную от правой части, т.е.

$$f'(\alpha) = \alpha\left(\vec{r}_k, A\vec{r}_k\right) - \left(\vec{r}_k, \left(\vec{b} - A\vec{x}_k\right)\right) = 0.$$

Тогда

$$\alpha_k = \frac{\left(\vec{r}_k, \vec{r}_k\right)}{\left(\vec{r}_k, A\vec{r}_k\right)}. \tag{4.17}$$

Псевдокод метода наискорейшего спуска можно представить следующим образом:

- ***выбрать*** $\vec{x}_0$***, вычислить*** $\vec{r}_0 = \vec{b} - A\vec{x}_0$***,***
- ***повторять для*** $k = 0,1,2,...$

$$\vec{z}_k = A\vec{r}_k\,, \tag{4.18}$$

$$\alpha_k = \frac{(\vec{r}_k, \vec{r}_k)}{(\vec{r}_k, \vec{z}_k)}\,, \tag{4.19}$$

$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{r}_k\,, \tag{4.20}$$

$$\vec{r}_{k+1} = \vec{r}_k - \alpha_k \vec{z}_k\,, \tag{4.21}$$

- ***рассчитать*** $(\vec{r}_{k+1}, \vec{r}_{k+1})$ ***и***

если $\sqrt{(\vec{r}_{k+1}, \vec{r}_{k+1})} \geq \varepsilon$***, то продолжать***, (4.22)

- ***конец цикла.***

Из формул (4.18) – (4.22) видно, что на каждой итерации метода производится одно умножение матрицы на вектор в (4.18), два скалярных произведения в (4.19) и (4.22), две линейные комбинации векторов **saxpy** ((4.20), (4.21)), а также несколько скалярных операций. При $n \gg 1$ временные затраты на проведение одной итерации метода можно оценить как

$$T_1 \approx \left(2 \cdot n^2 + 2 \cdot (2 \cdot n) + 2 \cdot (2 \cdot n)\right) \cdot t_a \approx 2 \cdot n^2 \cdot t_a\,. \tag{4.23}$$

Таким образом, по вычислительной трудоемкости при $n \gg 1$ выполнение одной итерации метода наискорейшего спуска аналогично выполнению итерации метода Якоби.

В параллельной реализации методов минимальных невязок или наискорейшего спуска как данные (векторы $\vec{b}, \vec{r}_k, \vec{x}_k$ и матрица A), так и основные операции (скалярные произведения, линейные комбинации векторов **saxpy**, умножение матрицы на вектор) разделяются на p используемых укрупненных подзадач. В дополнение к межпроцессорным обменам данных, требуемым для осуществления этих основных операций, необходимы также дополнительные коммуникации для проверки сходимости вычислительного процесса (выпол-

нение редукции **sum** или **max** при вычислении норм $\|\vec{r}_k\|$ и $\|\vec{x}_{k+1} - \vec{x}_k\|$).

Декомпозиция векторов осуществляется равномерно между p подзадачами (процессорными элементами), причем каждая подзадача содержит компоненты векторов с одним и тем же количеством индексных значений, например, на ПЭ**μ** $(\mu = 1,..., p)$ распределяются компоненты векторов $\vec{b}$, $\vec{r}_k$, $\vec{x}_k$ с номерами индексов $i = 1 + (\mu - 1) \times n / p,..., \mu \times n / p$, $(\mu = 1,..., p)$. Поэтому при выполнении операции линейной комбинации векторов **saxpy** не требуется межпроцессорной передачи данных, в то время как при вычислении скалярных произведений они необходимы.

Декомпозиция матрицы A может осуществляться по строкам, по столбцам или на блоки. Для плотно заполненной матрицы декомпозиция на строчные блоки $A_\mu \in \mathbb{R}^{n/p \times n}$ $(\mu = 1,..., p)$ обеспечивает вычисление произведения матрицы на вектор $(\vec{z}_k)_\mu = A_\mu \vec{r}_k$ без межпроцессорных коммуникаций. В то же время коммуникации потребуются на следующем этапе, когда необходимо получить остальные компоненты произведения $\vec{z}_k = A\vec{r}_k$ для вычисления скалярного произведения $(\vec{r}_k, \vec{z}_k)$ в (4.19).

При распределении по процессорным элементам строчных блоков, составленных из *n/p* строк матрицы ***A***, и *n/p* компонент векторов $\vec{b}, \vec{r}_k, \vec{x}_k$ коммуникации требуются для операций (4.19) и (4.22).

При проведении оценок ускорения и эффективности построенного параллельного алгоритма нужно иметь в виду, что объем вычислительной работы, выполняемой всеми подзадачами, в сумме совпадает с объемом вычислений в последовательном алгоритме метода наискорейшего спуска. На шагах параллельного алгоритма (4.19) и (4.22) требуется сборка векторов $\vec{z}_k$ и $\vec{r}_{k+1}$ на каждом ПЭ (пересылка каждым ПЭ остальным по *n/p* чисел) с последующим вычислением скалярных произведений. И хотя в этом случае производится дублирование вычислений скалярных произведений, однако не требуется пересылки полученных результатов операции **dot**.

Другая стратегия организации межпроцессорных коммуникаций для выполнения параллельного алгоритма заключается в следующем.

Для осуществления операции $\vec{z}_k = A\vec{r}_k$ на шаге (4.18) производится сборка вектора $\vec{r}_k$ на каждом ПЭ. В вычислении скалярного произведения участвуют все ПЭ. Рассчитав $\left[\left(\vec{r}_k, \vec{z}_k\right)\right]_{\text{ц}}$, полученное значение передается остальным ПЭ для проведения операции редукции **sum**.

В итоге, в первом варианте межпроцессорной передачи данных на каждой итерации требуется каждому ПЭ пересылать по *3n/p* чисел остальным, а во втором – по *n/p+2*, причем нет дублирования вычислений при вычислении **dot**.

Оценим временные затраты параллельного алгоритма на каждой итерации

$$T_p \approx 2\frac{n^2}{p}t_a + (p-1)\left(\frac{n}{p}+2\right)t_{comm}.$$

Тогда при $n/p \gg 1$

$$S_p = \frac{T_1}{T_p} \approx \frac{p}{1+\dfrac{(p-1)\kappa}{2n}}; \kappa = \frac{t_{comm}}{t_a} \gg 1.$$

Из полученных оценок видно, что ускорение параллельного метода наискорейшего спуска совпадает с ускорением метода Якоби. Аналогичные результаты могут быть получены и для построения параллельной версии метода минимальных невязок.

5. ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ СПЛАЙНОВ

В различных алгоритмах вычислительной математики часто возникает необходимость решения следующих задач: приближенного восстановления сеточной функции, приближенного представления функций, сглаживания функций, описания форм сложных изделий и их изображения с помощью компьютерной графики, перестроение сеток в процессе счета.

Если на сетке $\omega: a = x_0 < \ldots < x_n = b$ заданы значения некоторой сеточной функции $f_0, f_1, \ldots, f_n$, то интерполяционный многочлен Лагранжа для $f(x)$ решает задачу интерполирования на $[a,b]$. Однако практическое применение таких многочленов при больших n для функций с особенностями и малой гладкостью ограничено. Этот недостаток можно устранить, если использовать полиномиальные сплайны (***spline*** – упругий стержень, рейка).

Простейшим примером полиномиального сплайна первой степени является ломаная, «склеенная» во внутренних узлах сетки ω из линейных функций. Наиболее употребительными в приложениях в настоящее время стали кубические и бикубические сплайны. Использование сплайн-функций для приближенного решения краевых задач способствовало развитию метода конечных элементов, получению разностных схем сплайновой интерполяции. Сплайновые методы успешно применяются при решении двумерных задач интерполяции и сглаживания функций. Этот подход оказывает свое плодотворное влияние и на создание параллельных вычислительных алгоритмов суперкомпьютерной математики для решения сложных задач науки и техники.

5.1. Построение кубического сплайна

Пусть на отрезке $[a,b]$ вещественной оси Ox на сетке

$$\omega: a = x_0 < x_1 < \ldots < x_n = b, \quad h_i = x_i - x_{i-1}, \quad i = \overline{1,n},$$

заданы значения некоторой сеточной функции f_i, $i=0,1,\ldots,n$.

Для интерполяционного кубического сплайна $S(x)$ дефекта 1 имеют место следующие соотношения:

$$S(x)=\sum_{k=0}^{3}a_k^i(x-x_i)^k\ ,\ x\in[x_i,x_{i+1}]\ ,\ i=\overline{0,n-1}\,. \tag{5.1}$$

$$S(x_i)=f_i,\ i=\overline{0,n}. \tag{5.2}$$

$$S(x)\in C^2[a,b]\ , \tag{5.3}$$

где верхний индекс i обозначает номер звена сплайна.

Из (5.1) следует, что на ω сплайн *S(x)* зависит от *4n* коэффициентов

$$a_k^i\ ,i=\overline{0,n-1}\ ,\ k=\overline{0,3}.$$

Для их определения имеется *3(n –1)* условий непрерывности (5.3) во внутренних узлах сетки ω, а также *n +1* условие интерполяции (5.2). Таким образом, для определения коэффициентов сплайна необходимо задать еще два условия. Обычно рассматриваются четыре типа дополнительных условий:

$$S'(a)=f'(a),\ S'(b)=f'(b)\,. \tag{5.4}$$

$$S''(a)=f''(a),\ S''(b)=f''(b)\,. \tag{5.5}$$

$$S'(a)=S'(b),\ S''(a)=S''(b)\,. \tag{5.6}$$

$$S'''(x_1-0)=S'''(x_1+0),\ S'''(x_{n-1}-0)=S'''(x_{n-1}+0). \tag{5.7}$$

Дополнительные условия (5.6) используются для периодических сплайнов с периодом *(b–a)*. Если для непериодического сплайна отсутствует дополнительная информация о значении его первой или второй производной при *x=a* или *x=b*, то рекомендуется использовать дополнительные условия (5.7). В этом случае

$$a_k^0=a_k^1,a_k^{n-1}=a_k^{n-2},k=\overline{0,3}\ .$$

Введём обозначения:

$$M_i=S''(x_i),\ i=\overline{0,n}\,.$$

Величины M_i называются моментами сплайна *S(x)*.

Как следует из определения кубического сплайна, его вторая производная является линейной функцией на каждом элементарном интервале сетки ω. Для звена такой ломаной имеет место формула

$$S''(x)=\frac{x_i-x}{h_i}M_{i-1}+\frac{x-x_{i-1}}{h_i}M_i,\ h_i=x_i-x_{i-1}\ ,\ x\in[x_{i-1},x_i]. \quad (5.8)$$

Интегрируя (5.8) дважды по x, последовательно получим

$$S'(x)=-\frac{(x_i-x)^2}{2h_i}M_{i-1}+\frac{(x-x_{i-1})^2}{2h_i}M_i+c_1^{i-1}, \quad (5.9)$$

$$S(x)=\frac{(x_i-x)^3}{6h_i}M_{i-1}+\frac{(x-x_{i-1})^3}{6h_i}M_i+c_1^{i-1}x+c_2^{i-1}, \quad (5.10)$$

где c_1^{i-1}, c_2^{i-1} – константы интегрирования для $(i-1)$ интервала сетки ω.

Подставив (5.10) и условия интерполирования (5.2), получим

$$S'(x)=-\frac{(x_i-x)^2}{2h_i}M_{i-1}+\frac{(x-x_{i-1})^2}{2h_i}M_i+\frac{f_i-f_{i-1}}{h_i}-\frac{h_i}{6}(M_i-M_{i-1}), \quad (5.11)$$

$$S(x)=\frac{(x_i-x)^3}{6h_i}M_{i-1}+\frac{(x-x_{i-1})^3}{6h_i}M_i+(f_{i-1}-\frac{h_i^2}{6}M_{i-1})\frac{x_i-x}{h_i}+$$

$$+(f_i-\frac{h_i^2}{6}M_i)\frac{x-x_{i-1}}{h_i}\ ,\ x\in[x_{i-1},x_i]. \quad (5.12)$$

Отметим, что (5.11), (5.12) есть представление кубического сплайна и его первой производной на $(i-1)$ интервале сетки ω. На этом интервале $S''(x)$ определяется функцией (5.8), а

$$S'''(x)=\frac{1}{h_i}(M_i-M_{i-1})\ ,\ x\in[x_{i-1},x_i].$$

Если в выражениях (5.11), (5.12) заменить индекс i на $i+1$, то получим представление $S(x)$, $S'(x)$ на i-м интервале сетки ω:

$$S'(x)=-\frac{(x_{i+1}-x)^2}{2h_{i+1}}M_i+\frac{(x-x_i)^2}{2h_{i+1}}M_{i+1}+$$
$$+\frac{f_{i+1}-f_i}{h_{i+1}}-\frac{h_{i+1}}{6}(M_{i+1}-M_i), \quad (5.13)$$

$$S(x)=\frac{(x_{i+1}-x)^3}{6h_{i+1}}M_i+\frac{(x-x_i)^3}{6h_{i+1}}M_{i+1}+(f_i-\frac{h_{i+1}^2}{6}M_i)\frac{x_{i+1}-x}{h_{i+1}}+$$

$$+(f_{i+1}-\frac{h_{i+1}^2}{6}M_{i+1})\frac{x-x_i}{h_{i+1}},\ x\in[x_i,x_{i+1}]. \qquad (5.14)$$

Из (5.11) и (5.13) для $x=x_i$ запишем односторонние пределы производной:

$$S'(x_i-0)=\frac{h_i}{6}M_{i-1}+\frac{h_i}{3}M_i+\frac{f_i-f_{i-1}}{h_i}, \qquad (5.15)$$

$$S'(x_i+0)=-\frac{h_{i+1}}{6}M_{i+1}-\frac{h_{i+1}}{3}M_i+\frac{f_{i+1}-f_i}{h_{i+1}}. \qquad (5.16)$$

Из условия непрерывности первой производной сплайна во внутренних узлах сетки ω получим систему уравнений для определения моментов кубического сплайна, например, в случае дополнительных условий 1-го типа

$$2M_0+M_1=-\frac{6}{h_1}\left[f'(a)-\frac{f_1-f_0}{h_1}\right],$$

$$\alpha_i M_{i-1}+2M_i+\beta_i M_{i+1}=3f_{\overline{x}x},\ i=\overline{1,n-1}, \qquad (5.17)$$

$$2M_n+M_{n-1}=\frac{6}{h_n}\left[f'(b)-\frac{f_n-f_{n-1}}{h_n}\right],$$

где

$$\overline{h}_i=\frac{h_i+h_{i+1}}{2},\ \alpha_i=\frac{h_i}{2\overline{h}_i},\ \beta_i=\frac{h_{i+1}}{2\overline{h}_i},\ f_{\overline{x}x}=\frac{1}{\overline{h}_i}\left[\frac{f_{i+1}-f_i}{h_{i+1}}-\frac{f_i-f_{i-1}}{h_i}\right].$$

Определив методом прогонки из этой системы массив моментов и подставив M_i, M_{i+1}, f_i, f_{i+1} в выражение (5.14), получим представление $S(x)$ для $x\in[x_i,x_{i+1}]$. Отметим, что если при вычислении интегра-

ла $I = \int_a^b f(x)dx$ подынтегральную функцию *f(x)* заменить кубическим сплайном *S(x)*, то получим формулу численного интегрирования

$$\int_a^b S(x)dx = \frac{1}{2}\sum_{i=0}^{n-1} h_{i+1}(f_i + f_{i+1}) - \frac{1}{24}\sum_{i=0}^{n-1} h_{i+1}^3 (M_i + M_{i+1}),$$

которая имеет более высокую точность, чем формула трапеций.

Введём теперь в рассмотрение расширенную сетку

$$\overline{\omega} : x_{-3} < x_{-2} < \ldots < x_0 < \ldots < x_n < x_{n+1} < \ldots < x_{n+3}.$$

и нормализованные базисные функции $B_i(x)$, $i=-1,0,\ldots,n,n+1$, где x_i – центр носителя. Учитывая, что эти функции образуют базис в пространстве кубических сплайнов дефекта 1, сплайн $S(x)$ из этого множества можно представить в виде

$$S(x) = \sum_{i=-1}^{n+1} b_i B_i(x), \qquad (5.18)$$

где коэффициенты b_i подлежат определению.

Тогда, например, система уравнений для определения коэффициентов интерполяционного сплайна (5.18) для дополнительных условий 1-го типа (5.4) имеет вид

$$\begin{cases} b_{-1}B'_{-1}(x_0) + b_0 B'_0(x_0) + b_1 B'_1(x_0) = f_0', \\ b_{i-1}B_{i-1}(x_i) + b_i B_i(x_i) + b_{i+1}B_{i+1}(x_i) = f_i, \quad i = \overline{0,n}, \\ b_{n-1}B'_{n-1}(x_n) + b_n B'_n(x_n) + b_{n+1}B'_{n+1}(x_n) = f_n'. \end{cases} \qquad (5.19)$$

После исключения b_{-1} и b_{n+1} такую систему можно разрешить методом прогонки, если

$$\max_{\underbrace{|i-j|=1}} \frac{h_i}{h_j} < (1+\sqrt{13})/2 \approx 2.3,$$

что соответствует условию диагонального преобладания матрицы.

При вычислении $B_i(x)$ для $x \in (x_i, x_{i+1})$ можно использовать формулы:

$$B_{i-1}(x) = \frac{1}{6}(1-t)^3,\ B_i(x) = \frac{1}{6}\left[1 + 3(1-t) + 3t(1-t)^2\right],$$

$$B_{i+1}(x) = \frac{1}{6}\left[1 + 3t + 3t^2(1-t)\right],\quad B_{i+2}(x) = \frac{1}{6}t^3,$$

$$t = \frac{x - x_i}{h},\ t \in [0, 1], h = (b-a)/n.$$

Отметим, что представление сплайна через *B*-функции (5.18) удобно для больших *n*, так как для его вычисления требуется хранение двух массивов информации (узлов x_i и коэффициентов b_i). С другой стороны, изменение одного из коэффициентов в (5.18) влечёт изменение $S(x)$ лишь на носителе соответствующей *B*-функции, что можно использовать для локального изменения сплайна.

Значительное число прикладных задач связано с нахождением собственных чисел матриц большой размерности. Пусть $A = \left[a_{ij}\right]_1^n$ – вещественная симметричная матрица. Тогда, пользуясь теоремой Гершгорина, можно определить промежуток [*a,b*], содержащий все действительные собственные числа λ_i этой матрицы. Если на [*a,b*] ввести равномерную сетку ω и поставить ей в соответствие сеточную функцию

$$f_i = \det(A - \xi_i E),\ \xi_i = a + i \cdot h,\ h = (b-a)/n,\ i = \overline{0,n},$$

то задача об определении собственных чисел сплайновым методом сводится к нахождению корней кубического сплайна (полинома третьей степени) на каждом интервале сетки ω. Используя декомпозицию сеточной области по числу процессорных элементов p, анализ корней кубических уравнений можно осуществить в параллельном режиме после загрузки в память процессоров соответствующих значений M_i и f_i. Для контроля вычислений можно использовать известные соотношения

$$Sp(A) = \sum_{i=1}^{n} a_{ii} = \sum_{i=1}^{n} \lambda_i\ ,\ \det(A) = \pm\prod_{i=1}^{n} \lambda_i\ .$$

Отметим, что при определении действительных корней уравнения с помощью кубического сплайна шаг сетки h должен быть подобран так, чтобы на каждом интервале было не более трех корней.

5.2. Параллельный алгоритм построения кубического сплайна

Допустим, что на равномерной сетке ω: $a = x_0 < ... < x_n = b$ заданы значения некоторой сеточной функции $f_0, f_1, ..., f_n$ и $n \gg p \gg 1$. Рассмотрим один из вариантов параллельного алгоритма построения кубического сплайна с погрешностью $O(h^2)$.

Пусть $n = p \times q$, $q>3$, p – число процессоров, в памяти которых размещены соответственно данные:

$$M_0, f_0, f_1, ..., f_q, f_{q+1}; f_{q-1}, f_q, f_{q+1}, ..., f_{2q}, f_{2q+1}; ...;$$

$$f_{n-q-1}, f_{n-q}, f_{n-q+1}, ..., f_{n-1}, f_n, M_n$$

Такое расположение данных соответствует декомпозиции области ω с перекрытием на два шага сетки, фрагмент которой изображен на рис. 5.1.

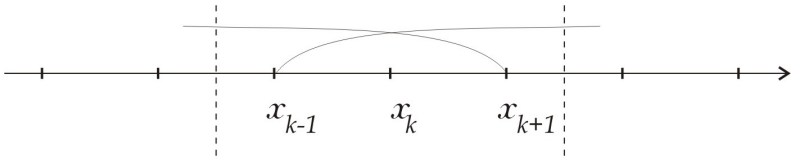


Рис. 5.1. Декомпозиция области ω с перекрытием

Процесс построения сплайна состоит из трех этапов:

1. Распределение исходных данных в каждом процессоре в соответствии с декомпозицией сетки ω.

2. Вычисление по сеточной функции с погрешностью $O(h^2)$ моментов сплайна на границах интервалов декомпозиции области:

$$M_k = (f_{k-1} - 2f_k + f_{k+1}) / h^2,$$
$$M_{k+q} = (f_{k+q-1} - 2f_{k+q} + f_{k+q+1}) / h^2, \qquad (5.20)$$
$$k = q, 2q, 3q, \cdots, (p-1)q.$$

3. Решение методом прогонки в каждом процессоре трехдиагональной системы для определения моментов. В этом случае потребу-

ется выполнить порядка *O(q)* операций на каждом процессоре. Степень параллелизма можно увеличить, используя один из методов распараллеливания метода прогонки.

Для вычисления значения сплайна в некоторой точке x из $[a,b]$ необходимо определить сеточный интервал, которому принадлежит эта точка, и воспользоваться формулой (5.12) или (5.14). Определение сеточного интервала можно осуществить параллельно, используя декомпозицию сеточной области без перекрытия.

Если сплайн восстанавливается через базисные функции, то к системе (5.19) на границах интервалов декомпозиции необходимо добавить уравнения

$$\begin{cases} b_{k-1}B''_{k-1}(x_k) + b_k B''_k(x_k) + b_{k+1}B''_{k+1}(x_k) = (f_{k-1} - 2f_k + f_{k+1})/h^2; \\ b_{k+q-1}B''_{k+q-1}(x_{k+q}) + b_{k+q} B''_{k+q}(x_{k+q}) + b_{k+q+1}B''_{k+q+1}(x_{k+q}) = \\ = (f_{k+q-1} - 2f_{k+q} + f_{k+q+1})/h^2; \\ k = q, 2q, 3q, \cdots, (p-1)q. \end{cases}$$

После приведения этой системы к трехдиагональному виду решение находится методом прогонки.

Учитывая свойства носителей B-сплайнов из (5.18), для вычисления $S(z)$, $z \in (x_i, x_{i+1}), i = \overline{0, n-1}$, удобно пользоваться формулой

$$S(z) = b_{i-1}B_{i-1}(z) + b_i B_i(z) + b_{i+1}B_{i+1}(z) + b_{i+2}B_{i+2}(z).$$

Вычисление сплайна по этой формуле можно осуществить на p-процессорной системе при декомпозиции сеточной области в соответствии с рис. 5.1.

Рассмотренный выше параллельный алгоритм построения кубического сплайна дефекта 1 при $p \to n$ приводит к построению локального кубического сплайна на каждом интервале сетки ω. Если не использовать условия (5.20), то решение сплайновых систем можно найти, например, методом циклической редукции из главы 3.

5.3. Сплайны двух переменных

Рассмотрим прямоугольную область

$$\Omega = \{x,y;\ \ 0 \le x \le a;\ \ 0 \le y \le b\},$$

изображенную на рис. 5.2, в которой введена сетка $\omega = \omega_x \times \omega_y$, где

$$\omega_x = \{x_i, 0 = x_0 < x_1 < \ldots < x_n = a\},$$
$$\omega_y = \{y_j, 0 = y_0 < y_1 < \ldots < y_m = b\}.$$

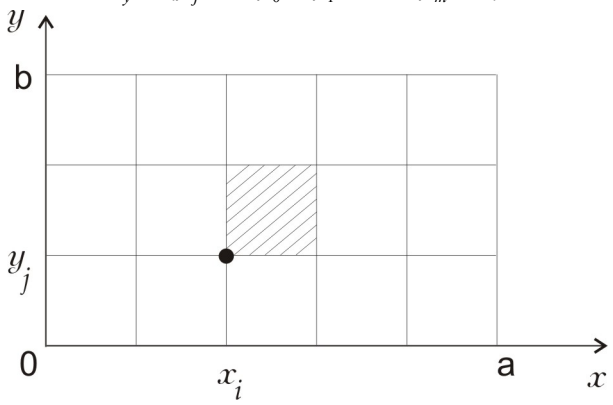


Рис. 5.2. Прямоугольная сетка с выделенной ячейкой

Пусть
$\Omega_{ij} = \{x_i \le x \le x_{i+1},\ y_j \le y \le y_{j+1},\ h_i = x_{i+1} - x_i,\ l_j = y_{j+1} - y_j\}$ – элементарная ячейка сетки, в узлах которой заданы значения сеточной функции $f_{i,j}$ ($i = \overline{0,n}, j = \overline{0,m}$). Задача интерполирования состоит в восстановлении этой функции в любой точке области Ω.

Билинейный интерполяционный сплайн для каждой ячейки Ω_{ij} можно записать и в виде

$$S(x, y) = a_{ij}x + b_{ij}y + c_{ij}xy + d_{ij}. \qquad (5.21)$$

Для определения четырех коэффициентов имеется четыре условия интерполяции в вершинах ячейки сетки Ω_{ij}:

$$S(x_i, y_j) = f_{ij},\ S(x_{i+1}, y_j) = f_{i+1\,j},$$
$$S(x_i, y_{j+1}) = f_{ij+1},\ S(x_{i+1}, y_{j+1}) = f_{i+1\,j+1}.$$

Тогда в (5.21)

$$a_{ij} = \frac{1}{l_j h_i}[(f_{i+1j} - f_{ij})y_{j+1} + (f_{ij+1} - f_{i+1j+1})y_j],$$

$$b_{ij} = \frac{1}{l_j h_i}[(f_{ij+1} - f_{ij})x_{i+1} + (f_{i+1j} - f_{i+1j+1})x_i],$$

$$c_{ij} = \frac{1}{l_j h_i}[f_{ij} - f_{i+1j} - f_{ij+1} + f_{i+1j+1}],$$

$$d_{ij} = \frac{1}{l_j h_i}[f_{ij}x_{i+1}y_{j+1} - f_{i+1j}x_i y_{j+1} - f_{ij+1}x_{i+1}y_j + f_{i+1j+1}x_i y_j].$$

Параллельный алгоритм вычисления коэффициентов такого сплайна на p процессорных элементах легко реализуется декомпозицией рассматриваемой области на p прямоугольников. Отметим, что аналогичным образом можно вычислить интеграл по области Ω от заданной сеточной функции на p процессорах, так как для Ω_{ij}

$$I_{ij} = \iint\limits_{\Omega_{ij}} S(x,y)\,dxdy = \frac{1}{4}(f_{ij} + f_{i+1j} + f_{ij+1} + f_{i+1j+1})h_i l_j .$$

Билинейные интерполяционные сплайны удобно использовать для восстановления непрерывных функций. Если возникает необходимость восстановления в любой точке области Ω не только самой функции, но и ее первых и вторых производных, то эту задачу можно решить в классе бикубических сплайнов.

Бикубический интерполяционный сплайн $S(x,y)$ дефекта 1 удовлетворяет следующим трем условиям:

$$S(x,y) = \sum_{k=0}^{3}\sum_{l=0}^{3} a_{kl}^{ij}(x - x_i)^k (y - y_j)^l,\ (x,y) \in \Omega_{ij} . \qquad (5.22)$$

$$S(x,y) \in C^{2,2}(\Omega) . \qquad (5.23)$$

$$S(x_i, y_j) = f_{ij},\ i = \overline{0,n},\ \ j = \overline{0,m} . \qquad (5.24)$$

Для однозначного нахождения такого сплайна необходимо задавать дополнительные условия в граничных точках сетки ω. Например, для

дополнительных условий типа (5.5) должны быть известны значения производных:

$$\frac{\partial^2 S(x_i, y_j)}{\partial x^2} = f_{ij}^{(xx)}, \qquad i = 0, n, \; j = \overline{0, m}\,, \tag{5.25}$$

$$\frac{\partial^2 S(x_i, y_j)}{\partial y^2} = f_{ij}^{(yy)}, \qquad i = \overline{0, n}, \; j = 0, m\,, \tag{5.26}$$

$$\frac{\partial^4 S(x_i, y_j)}{\partial x^2 \partial y^2} = f_{ij}^{(xxyy)}, \qquad i = 0, n, \; j = 0, m\,, \tag{5.27}$$

которые обеспечивают однозначное определение сплайна.

Рассмотрим алгоритм построения сплайна (5.22) через моменты, основанный на идее метода расщепления. Этот метод позволяет свести решение двумерной задачи к цепочке одномерных задач о восстановлении коэффициентов одномерных сплайнов. Введем обозначения:

$$M_{ij} = S_{xx}(x_i, y_j), \quad N_{ij} = S_{yy}(x_i, y_j), \quad K_{ij} = S_{xxyy}(x_i, y_j)\,.$$

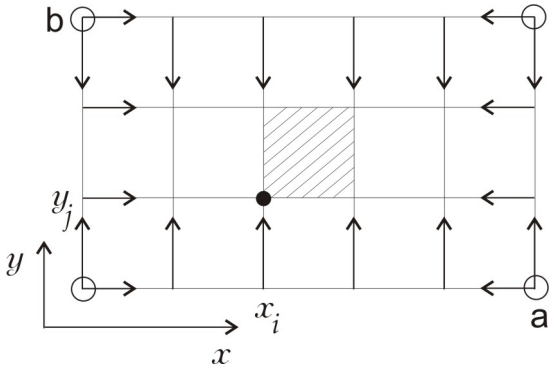


Рис. 5.3. Структура задания дополнительных условий для $S(x, y)$

Пусть известны дополнительные условия 2-го типа (5.25) – (5.27). Наглядное представление о задании таких условий в граничных узлах сетки дано на рис. 5.3.

Здесь стрелки соответствуют заданию вторых производных на границе рассматриваемой области Ω. В угловых точках, кроме вторых производных по x и y, задаются еще и значения (5.27).

Вычисление сплайна в точке *(x.y)* представим в виде четырех последовательных этапов.

I этап. Для каждого $i = \overline{0,n}$ построим одномерные сплайны по переменной y:

$$S(x_i, y) = \phi(N_{ij}, y).$$

При этом будут определены на сетке ω значения

$$N_{ij}, \quad i = \overline{0,n}, \quad j = \overline{0,m}.$$

На этом этапе каждый из p процессорных элементов после загрузки соответствующих данных вычисляет часть массива моментов методом прогонки на q сеточных линиях (лентах) по x, если $p \cdot q = n+1$.

II этап. Для $j = \overline{0,m}$ построим одномерные сплайны по переменной x:

$$S(x, y_j) = \psi(M_{ij}, x).$$

При этом будут определены на сетке ω значения

$$M_{ij}, \quad j = \overline{0,m}, \quad i = \overline{0,n}.$$

Здесь каждый из p процессорных элементов после загрузки соответствующих данных может находить массивы моментов методом прогонки на g сеточных линиях (лентах) по y, если $p \cdot g = m+1$.

III этап. По известной при $y = 0$ сеточной функции

$$\{f_{00}^{(yy)}, f_{10}^{(yy)}, \ldots, f_{n0}^{(yy)}\}$$

с дополнительными условиями

$$\frac{\partial^4 S(x_0, y_0)}{\partial x^2 \partial y^2} = f_{00}^{(xxyy)}, \qquad \frac{\partial^4 S(x_n, y_0)}{\partial x^2 \partial y^2} = f_{n0}^{(xxyy)},$$

построив сплайн через моменты по переменной x, получим вектор

$$K_0 = \{K_{00}, K_{10}, ..., K_{n0}\}.$$

Аналогичным образом вычислим при $y = b$ вектор

$$K_M = \{K_{0m}, K_{1m}, ..., K_{nm}\}.$$

Очевидно, что нахождение векторов $K_0, ..., K_n$ можно осуществить методом прогонки одновременно и независимо.

Учитывая эти векторы, по значениям сеточной функции M_{ij} для $i = \overline{0,n}$ решаются одномерные задачи об определении моментов сплайна по переменной y по аналогии с этапом II. Этот процесс дает возможность определить двумерный массив

$$K_{ij} = S_{xxyy}(x_i, y_j),\ i = \overline{0,n},\quad j = \overline{0,m}.$$

Каждый из p процессорных элементов после загрузки соответствующих данных может находить массивы моментов методом прогонки на q линиях по x, если $p \cdot q = n + 1$.

IV этап. После осуществления первых трех этапов, определив решение $2n + m + 5$ систем с трехдиагональными матрицами, получим двумерные массивы

$$f_{ij},\quad N_{ij},\quad M_{ij},\quad K_{ij},\quad i = \overline{0,n},\quad j = \overline{0,m}.$$

Тогда в угловых точках любой клетки Ω_{ij} можно записать по 4 условия на сплайн и его производные, что дает возможность определить 16 коэффициентов из (5.22) в клетке Ω_{ij}. Для вычисления сплайна в некоторой точке $P(x, y)$ необходимо вначале осуществить поиск клетки, которой она принадлежит, а затем вычислить $S(x, y)$ по формуле (5.22). При этом в памяти компьютера потребуется хранить $16mn$ чисел. Для их определения можно использовать декомпозицию области Ω на p прямоугольников. Для сокращения объема хранимой информации в 4 раза можно использовать вычисление сплайна непосредственно через элементы массивов

$$f_{ij},\quad N_{ij},\quad M_{ij},\quad K_{ij},\quad i = \overline{0,n},\quad j = \overline{0,m}.$$

Отметим, что на каждом этапе рассмотренного вычислительного алгоритма решаются трехдиагональные системы линейных уравнений, отличающиеся только правыми частями (за исключением определения векторов K_0, K_m). Тогда в любом из p процессорных элементов массив первых прогоночных коэффициентов на каждом этапе достаточно вычислять только один раз, что позволит реализовать метод прогонки за $5n$ или $5m$ операций.

Таким образом, нахождение моментов бикубического сплайна в однопроцессорном варианте требует порядка $27n \times m$ операций. С учетом хранения массива первых прогоночных коэффициентов требуется порядка $18n \times m$ операций. Реализация алгоритма на p-процессорной системе с распределенной памятью для нахождения моментов, например при $m = n$ приводит к необходимости дублирования порядка $9n \times p$ операций.

Рассмотренный выше параллельный алгоритм на каждом этапе основан на одновременном решении независимых трехдиагональных систем в каждом процессорном элементе. Степень параллелизма можно увеличить, если использовать метод циклической редукции (см. главу 3) или итерационные алгоритмы из главы 4.

6. ВЫЧИСЛЕНИЕ ОПРЕДЕЛЕННЫХ И КРАТНЫХ ИНТЕГРАЛОВ

Численное интегрирование – это приближенное вычисление значения определенного интеграла с помощью численных методов. Обычно численное интегрирование применяется, когда невозможно или весьма трудоемко аналитически найти точное значение интеграла либо когда подынтегральная функция задана таблично.

Задача вычисления определенных и кратных интегралов может быть разделена на несколько независимых подзадач путем разбиения области интегрирования на непересекающиеся подобласти, по которым проводится интегрирование, в том числе и с использованием численных методов. Такое свойство интегралов открывает большие возможности для параллельных вычислений.

6.1. Вычисление определенных интегралов

Пусть требуется найти значение определенного интеграла Римана

$$\int_a^b f(x)dx \tag{6.1}$$

для некоторой функции $f(x)$. Простые квадратурные формулы можно вывести непосредственно из определения интеграла, то есть из представления

$$I = \lim_{n\to\infty} \sum_{i=1}^{n} f(\xi_i) * (x_i - x_{i-1}), \tag{6.2}$$

где $\{x_i\}_{i=0}^{n}$ – произвольная упорядоченная система точек отрезка $[a,b]$ такая, что $\{x_0 - a, x_i - x_{i-1}, b - x_n\} \to 0$ при $n \to \infty$, а ξ_i – произвольная точка промежутка $[x_{i-1}, x_i]$. Зафиксировав некоторое $n \geq 1$, будем иметь

$$I \approx \sum_{i=1}^{n} f(\xi_i)(x_i - x_{i-1}). \tag{6.3}$$

Это приближенное равенство называется общей формулой прямоугольников. Площадь криволинейной трапеции приближенно заменяется площадью ступенчатой фигуры, составленной из прямоугольников, основаниями которых служат отрезки $[x_{i-1}, x_i]$, а высотами – ординаты $f(\xi_i)$.

Квадратурная формула средних прямоугольников

Чтобы из общей формулы (6.3) получить конструктивное правило приближенного вычисления интеграла, воспользуемся свободой расположения точек x_i, разбивающих промежуток интегрирования $[a,b]$ на элементарные отрезки $[x_{i-1}, x_i]$, и свободой выбора точек ξ_i на этих отрезках. Условимся пользоваться равномерным разбиением отрезка $[a,b]$ на n равных частей точками x_i с шагом $h = (b-a)/n$, полагая

$$x_0 = a,\ \ x_i = x_{i-1} + h,\ \ (i = 1,2,...n-1),\ \ x_n = b\,. \qquad (6.4)$$

При таком разбиении формула (6.3) приобретает вид

$$I \approx h\sum_{i=1}^{n} f(\xi_i)\,,\ \ \xi \in [x_{i-1}, x_i]\,. \qquad (6.5)$$

Рассмотрим три случая фиксирования точек ξ_i на элементарных отрезках $[x_{i-1}, x_i]$.

1. Положим $\xi_1 = x_{i-1}$. Тогда из (6.5) получим

$$I \approx I^{П-} = h\sum_{i=1}^{n} f(x_{i-1})\,. \qquad (6.6)$$

2. Пусть в (6.5) $\xi_i = x_i$. Тогда имеем

$$I \approx I^{П+} = h\sum_{i=1}^{n} f(x_i)\ . \qquad (6.7)$$

Формулы (6.6) и (6.7) называются квадратурными формулами левых и правых прямоугольников соответственно. Очевидно, что $I^{П-}$ и $I^{П+}$ дают двусторонние приближения к значению I интеграла от

монотонной функции. Можно рассчитывать на большую точность получения значения интеграла, если взять точку посередине между точками x_{i-1} и x_i. Отсюда приходим к следующему случаю.

3. Фиксируем $\xi_i = \frac{1}{2}(x_{i-1} + x_i)$ $\left\{ = x_{i-1} + \frac{h}{2} = x_i - \frac{h}{2} \right\}$.

В результате получим квадратурную формулу средних прямоугольников (рис. 6.1):

$$I \approx I^{П} = h\sum_{i=1}^{n} f\left(x_{i-1} + \frac{h}{2}\right) = h\sum_{i=1}^{n} f\left(x_i - \frac{h}{2}\right). \qquad (6.8)$$

При проведении вычислений по этой формуле не используются значения функции на концах промежутка интегрирования.

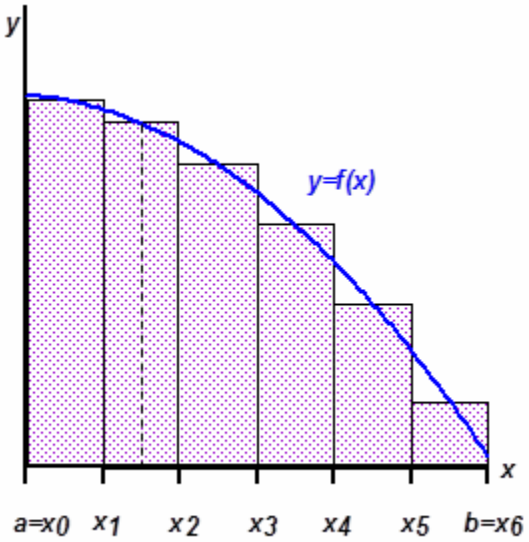


Рис. 6.1. Графическая интерпретация метода средних прямоугольников ($n = 6$)

Погрешность метода средних прямоугольников может быть оценена по следующей формуле:

$$r^{П} = I - I^{П} = \frac{b-a}{24} h^2 f''(\xi_T), \xi_T \in (a,b).$$

Квадратурная формула трапеций

Простейшая формула трапеций с остаточным членом применительно к интегрированию по отрезку $[x_{i-1}, x_i]$ может быть записана в виде точного равенства:

$$\int_{x_{i-1}}^{x_i} f(x)dx = \frac{f_{i-1}+f_i}{2}h - \frac{f''(\xi_i)}{12}h^3, \qquad (6.9)$$

где ξ_i – некоторая точка интервала (x_{i-1}, x_i), а $f_i = f(x_i)$. Выполняя разбиение исходного промежутка интегрирования $[a,b]$ на n частей с шагом $h = (b-a)/n$ и применяя к каждой части формулу (6.9), имеем

$$\int_a^b f(x)dx = \sum_{i=1}^{n}\int_{x_{i-1}}^{x_i} f(x)dx = \frac{h}{2}\sum_{i=1}^{n}(f_{i-1}+f_i) - \frac{h^3}{12}\sum_{i=1}^{n} f''(\xi_i). \qquad (6.10)$$

Отсюда следует, что искомое значение интеграла можно приближенно найти по формуле

$$I \approx I^T = h\left(\frac{f_0+f_n}{2} + f_1 + f_2 + \ldots + f_{n-1}\right), \qquad (6.11)$$

которая называется формулой трапеций, а погрешность формулы (6.11) можно охарактеризовать остаточным членом r^T, полученным упрощением последнего слагаемого в правой части (6.10).

По обобщенной теореме о среднем значении функции на отрезке существует точка $\xi_T \in (a,b)$, такая, что $\sum_{i=1}^{n} hf''(\xi_i) = f''(\xi_T)(b-a)$ (площадь ступенчатой фигуры, составленной из прямоугольников с основаниями h и высотами $f''(\xi_i)$, можно отождествить с площадью одного прямоугольника с основанием $\sum_{i=1}^{n} h = b-a$ и высотой $f''(\xi_T)$). Таким образом, остаточный член формулы трапеций (6.11) есть

$$r^T = I - I^T = -\frac{b-a}{12}h^2 f''(\xi_T), \xi_T \in (a,b). \qquad (6.12)$$

Остаточный член формулы средних прямоугольников примерно вдвое меньше, чем у формулы трапеций, следовательно, формула средних точнее. Знаки главного члена погрешности у формулы трапеций и средних разные.

Адаптивные алгоритмы численного интегрирования

Точность получаемых результатов в методах численного интегрирования зависит как от типа квадратурной формулы, характера изменения подынтегральной функции, так и от шага интегрирования. Если считать, что величина шага задана, то для достижения сравнимой точности при интегрировании слабо меняющейся функции шаг можно выбирать большим, чем при интегрировании резко меняющихся функций.

При решении прикладных задач часто встречаются случаи, когда поведение подынтегральной функции меняется на отдельных участках интегрирования. В связи с этим возникает необходимость строить экономичные численные алгоритмы, которые автоматически приспосабливались бы к характеру изменения функции. Такие алгоритмы называются адаптивными (приспосабливающимися). Они позволяют вводить разные значения шага интегрирования на отдельных участках отрезка интегрирования, что дает возможность уменьшить машинное время без потери точности результатов расчета.

Рассмотрим принцип работы адаптивного алгоритма. Отрезок $[a,b]$ разбиваем на n элементарных отрезков, которые на каждом шаге адаптивного алгоритма делим пополам и на получившейся более подробной сетке заново вычисляем приближенное значение интеграла. Окончательное число шагов половинного деления для отдельного элементарного отрезка зависит от вида подынтегральной функции и допустимой погрешности вычислений приближенного значения интеграла $\varepsilon > 0$.

К каждому элементарному отрезку $[x_{i-1}, x_i]$ применяем формулы численного интегрирования при двух различных его разбиениях. Получаем приближения $I_i^{(1)}, I_i^{(2)}$ для интеграла по этому отрезку:

$$I_i = \int_{x_{i-1}}^{x_i} f(x)dx\,, \tag{6.13}$$

которые сравниваем в сходственных точках и проводим оценку погрешности. В случае попадания погрешности в допустимые границы последнее из полученных приближений принимаем за значение интеграла по соответствующему элементарному отрезку, иначе производим дальнейшее деление отрезка и вычисление новых приближений. При этом точки деления располагаются таким образом, чтобы использовались уже вычисленные значения функции в точках предыдущего разбиения, что поможет сэкономить время вычислений.

Например, при вычислении интеграла по формуле трапеций получаем следующие выражения:

$$I_i^{(1)} = \frac{h_i}{2}\left(f_{i-1} + f_i\right),$$

$$I_i^{(2)} = \frac{1}{2}\frac{h_i}{2}\left(f_{i-1} + f_i\right) + f_{i-1/2}\frac{h_i}{2}.$$

Используя вычисленные значения функции в точках предыдущего разбиения, получаем

$$I_i^{(2)} = I_i^{(1)} / 2 + \frac{h_i}{2} f_{i-1/2}\,.$$

В общем случае для любого числа равномерных разбиений $m = 2^q, q = 2,3,...$ имеет место формула

$$I_i^{(q)} = I_i^{(q-1)} / 2 + \frac{h_i}{2^{q-1}} \sum_{k=0}^{m/4-1} f\left(x_{i-1} + \frac{(2k+1)h_i}{2^{q-1}}\right).$$

Процесс деления отрезка пополам и вычисления уточненных значений $I_i^{(q)}, I_i^{(q-1)}$ продолжается до тех пор, пока модуль их разности станет меньше некоторой заданной величины δ_i, зависящей от ε и h:

$$\left|I_i^{(q)} - I_i^{(q-1)}\right| \le \delta_i\,,\ i = 1,2,...,n\,.$$

Аналогичная процедура проводится для всех n элементарных отрезков. Величина $\tilde{I} = \sum_{i=1}^{n} I_i^{(q_i)}$ принимается в качестве искомого значения интеграла I.

Параллельная реализация квадратурных формул средних прямоугольников

Рассмотрим параллельную реализацию квадратурной формулы средних прямоугольников для вычисления значения определенного интеграла от функции $f(x)$.

1. *Декомпозиция.* Область интегрирования $[a,b]$ разобьем на подобласти, а исходный интеграл представим в виде суммы интегралов по таким частичным отрезкам.

2. *Проектирование коммуникаций.* Для каждой из подобластей с использованием выбранной квадратурной формулы производится приближенное вычисление интеграла. Для этого необходимо задать значения пределов интегрирования, число узлов квадратурной формулы (и если надо, то и табличные значения функции в этих узлах). После проведения расчетов полученные результаты требуется передать на один процессорный элемент, который осуществляет нахождение суммы интегралов по частичным отрезкам.

3. *Укрупнение.* Объединение подобластей, по которым будет приближенно вычисляться интеграл, проводится с учетом планируемого к использованию числа процессоров p, минимизации межпроцессорной передачи данных и равномерной загрузки процессоров.

4. *Планирование вычислений.* При использовании одинаковых процессорных элементов укрупненные подзадачи назначаются каждому ПЭ. Часто в таком случае целесообразно использовать стратегию «хозяин/работник» (см. рис. 0.3 Введения), когда один ПЭ («хозяин») распределяет вычислительную нагрузку среди остальных ПЭ и после расчетов собирает от них полученные результаты для вычисления приближенного значения интеграла.

В случае разбиения отрезка $[a,b]$ на подобласти по числу используемых процессорных элементов p формула средних прямоугольников примет вид ($h = (b-a)/n$)

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f_{i-1/2}h = \\ = h\sum_{\mu=0}^{p-1}\left[\sum_{i=1}^{n/p} f(0.5(x_{i-1+\mu(n/p)} + x_{i+\mu(n/p)}))\right]. \qquad (6.14)$$

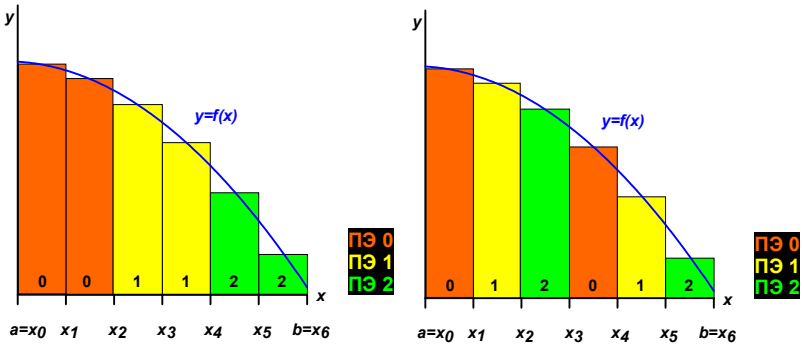


Рис. 6.2. Блочная (слева) и циклическая (справа) схемы распределения подобластей по процессорным элементам (ПЭ)

При этом схема вычислений состоит из следующих этапов: во-первых, на каждом процессорном элементе определяются границы интегрирования и число интервалов разбиения (n/p в полученных формулах), приходящихся на данную подобласть.

Во-вторых, каждый процессорный элемент вычисляет интеграл по квадратурной формуле по своему частичному отрезку (рис. 6.2, блочная схема). В (6.14) суммы в квадратных скобках могут вычисляться одновременно и независимо. На третьем этапе происходит передача всех вычисленных значений выбранному процессорному элементу и их суммирование для получения приближенного значения интеграла.

Отметим, что наряду с блочным распределением вычислений по процессорам можно использовать и другие способы, например циклическую схему (рис. 6.2) или схему с отражениями.

При использовании адаптивных алгоритмов численного интегрирования подход распараллеливания несколько меняется. Полученные после этапов декомпозиции, проектирования коммуникаций и ук-

рупнения подзадачи могут отличаться по объему вычислительной работы из-за итерационной схемы адаптивного алгоритма. Поэтому целесообразно вместо статического (предопределенного заранее) распределения подзадач по процессорным элементам использовать динамическую схему «хозяин/работник», при которой один процессорный элемент («хозяин») распределяет вычислительную работу между остальными, причем в силу различной трудоемкости вычислительных подзадач по времени их выполнения один ПЭ может закончить вычисления раньше других и главный процессорный элемент («хозяин») должен обеспечить его подзадачей из оставшегося объема вычислительных задач.

Заметим, что используя соответствующую линейную комбинацию уже вычисленных значений интеграла на последовательности сгущающихся сеток, можно повысить точность вычисления интеграла, например, по формулам Ричардсона.

6.2. Вычисление кратных интегралов

В качестве методов приближенного вычисления многомерных интегралов рассмотрим метод ячеек, метод повторного интегрирования и метод статистических испытаний.

Метод ячеек

Рассмотрим интеграл от функции $f(x,y)$ по прямоугольнику $D=\{a\le x\le b, c\le y\le d\}$:

$$I=\iint\limits_D f(x,y)dxdy\,.$$

По аналогии с формулой средних прямоугольников в одномерном случае можно приближенно вычислить этот интеграл как

$$\int\limits_c^d\int\limits_a^b f(x,y)dxdy\approx f\left(0{,}5(b+a),0{,}5(d+c)\right)(b-a)(d-c)\,.$$

Для повышения точности можно разбить прямоугольник на n непересекающихся подобластей прямоугольной формы и применять указанную выше формулу для каждой из них. Тогда

$$I \approx \sum_{i=1}^{n} f(x_i, y_i) S_i \ . \qquad (6.15)$$

Здесь n – количество подобластей; x_i, y_i, S_i – координаты их центров и площадь. Эта формула имеет второй порядок точности, как и составная формула средних прямоугольников в одномерном случае.

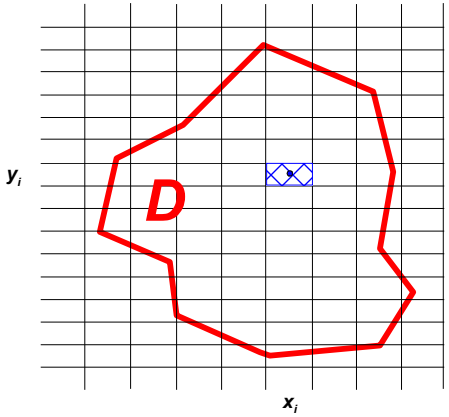


Рис. 6.3. Построение сетки для области
с криволинейной границей

Для области D с криволинейной границей метод ячеек строится следующим образом (рис. 6.3). Область D заключается в прямоугольник Ω, который покрывается сеткой с прямоугольными ячейками. Для вычисления интеграла во внутренних ячейках (все точки которых принадлежат области D) применяется формула (6.15). В граничных ячейках для вычисления интеграла используются некоторые дополнительные соглашения. Например, площадь треугольника или трапеции, одна граница которых является криволинейной, заменяется площадью соответствующей фигуры со спрямленной границей, а значение подынтегральной функции вычисляется в центре масс фигуры со спрямленной границей. Можно, снижая трудоемкость вычислений, вообще не включать в вычисляемое приближенное значение интегралов граничные ячейки. Но в этом случае следует позаботиться об увеличении количества ячеек сетки n, покрываю-

щей область D, поскольку в этом случае формула интегрирования (6.15) имеет лишь первый порядок точности.

Другим способом упрощения вычислений является замена подынтегральной функции $f(x,y)$ на функцию, определенную на области Ω – минимальном прямоугольнике, в который вписана область D:

$$\varphi(x,y)=\begin{cases} f(x,y),(x,y)\in D, \\ 0, \quad (x,y)\in \Omega \setminus D. \end{cases}$$

Тогда для области Ω вводится равномерная прямоугольная сетка и при вычислении интеграла используется формула (6.15).

Метод ячеек легко переносится на большее число измерений, однако с повышением кратности интеграла резко возрастает объем вычислений. Например, если интервал изменения каждой переменной разбить всего на сто частей, то для вычисления тройного интеграла потребуется вычислить сумму миллиона слагаемых. Параллельные вычисления по формуле (6.15) могут быть организованы с использованием декомпозиции области интегрирования.

Метод повторного интегрирования

При таком подходе в области D проводятся хорды, параллельные оси ***Ox***, и на них определенным образом вводятся узлы (рис. 6.4).

Двойной интеграл представляется в виде

$$I=\iint\limits_D f(x,y)dxdy=\int\limits_c^d F(y)dy; \;\; F(y)=\int\limits_{\gamma(y)}^{\chi(y)} f(x,y)dx\,.$$

Здесь $\gamma(y),\chi(y)$ – определенные на отрезке $[c,d]$ функции, представляющие левую и правую границы области D.

Сначала вычисляется интеграл по ***x*** вдоль каждой хорды по одномерной квадратурной формуле, затем интеграл по ***y*** – здесь в качестве узлов можно взять проекции хорд на ось ординат. Сведение кратного интеграла к повторному интегрированию по одномерным квадратурным формулам позволяет использовать все представленные выше вычислительные технологии для расчета определенных интегралов.

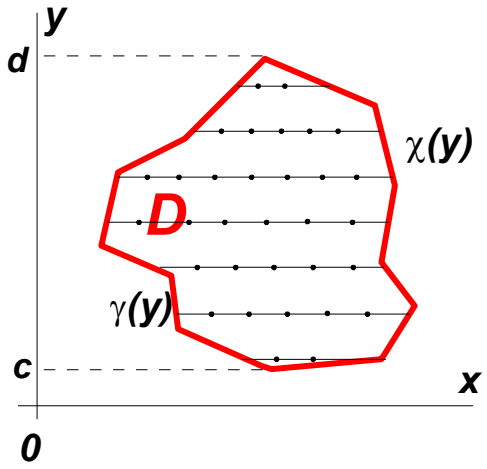


Рис. 6.4. Распределение узлов по области интегрирования
в методе повторного интегрирования

При построении параллельного метода вычисления приближенного значения интеграла в этом случае можно использовать мелкозернистый параллелизм при вычислении $F(y)$ вдоль соответствующей хорды или более крупнозернистый – распределив по несколько хорд на каждую укрупненную подзадачу.

Метод статистических испытаний

Метод Монте-Карло (метод статистических испытаний) для вычисления многомерных интегралов впервые был введен Ферми, а затем использовался фон Нейманом и Уламом в Лос-Аламосе в связи с моделированием нейтронной диффузии в расщепляемом материале. Этот способ часто используется для решения задач в различных областях физики, химии, математики, экономики, оптимизации, теории управления и др.

При вычислении приближенного значения интеграла

$$I = \iint_D f(x,y)dxdy$$

методом статистических испытаний сначала определяют границы прямоугольника $[a,b]\times[c,d]$, в который вписана область интегриро-

вания D. Затем с помощью датчика псевдослучайных чисел генерируется последовательность из n точек с координатами $(\xi_i, \eta_i), i = 1,...,n$, значения которых равномерно распределены на соответствующих отрезках $[a,b]$ и $[c,d]$. Для каждой точки проверяется факт ее попадания в область D, после чего вычисляется значение функции в соответствующей точке (рис. 6.5).

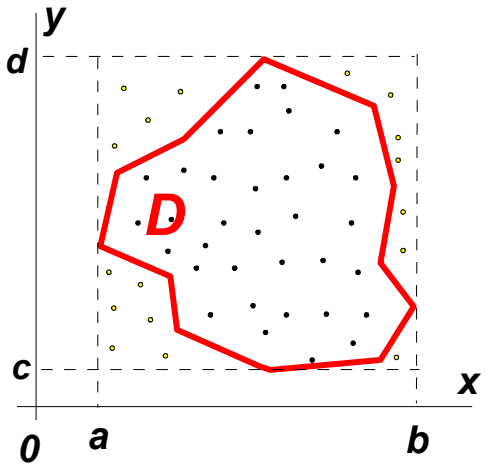


Рис. 6.5. Применение метода Монте-Карло
при вычислении двойного интеграла

Тогда формула метода статистических испытаний имеет вид

$$I \approx \frac{(b-a)(d-c)}{n} \sum_{i=1}^{n_{in}} f(\xi_i, \eta_i) .$$

Здесь n – количество испытаний, n_{in} – количество точек, попавших в область D.

Чем больше n, тем более точным будет результат. Метод обладает медленной сходимостью, так как его погрешность оценивается как $O(n^{-1/2})$, т.е. для увеличения точности вычислений в 10 раз нужно в 100 раз увеличить количество испытаний n. Однако независимость вычислений частичных сумм в методе Монте-Карло позволяет рассматривать его как метод с практически идеальным параллелизмом.

Отметим, что метод Монте-Карло можно эффективно использовать для вычисления площадей и объемов нерегулярных фигур.

Для использования метода Монте-Карло при вычислении интегралов, как и в других приложениях этого метода, необходимо вырабатывать последовательности случайных чисел с равномерным (или заданным по условиям моделирования) законом распределения вероятностей. Наиболее распространенный способ выработки случайных чисел на компьютере состоит в использовании специального алгоритма получения таких чисел. Поскольку эти числа генерируются по наперед заданному алгоритму, то получаемую последовательность можно воспроизвести сколько угодно раз. Тем не менее получающиеся псевдослучайные числа обладают всеми необходимыми статистическими свойствами, характерными для значений истинно случайной величины. Генератор псевдослучайных чисел должен удовлетворять определенным требованиям:

- псевдослучайные числа должны быть равномерно распределены;
- последовательности генерируемых псевдослучайных чисел не должны быть коррелированы;
- последовательности псевдослучайных чисел должны быть воспроизводимы, чтобы отлаживать программы и вычислительные модели;
- порядок генерации псевдослучайных чисел не должен зависеть от компьютера и программного обеспечения;
- период последовательности генерируемых псевдослучайных чисел должен быть на несколько порядков больше используемых в вычислениях подпоследовательностей;
- алгоритм генерации псевдослучайных чисел должен хорошо распараллеливаться.

Располагая удовлетворяющим указанным требованиям генератором псевдослучайных чисел, вычисление интеграла можно проводить одновременно и независимо несколькими процессорными элементами. Коммуникации потребуются только при получении итогового результата.

В настоящее время широко используется при применении параллельных методов Монте-Карло для решения математических задач библиотека генераторов псевдослучайных чисел для высокопроизводительных параллельных вычислительных систем SPRNG (Scalable, Portable, Random Number Generators). Имеющиеся в SPRNG генера-

торы удовлетворяют следующим требованиям: воспроизводимость, небольшие затраты времени на обмен данными, универсальность, качество, удобный интерфейс работы с программами, написанными на на языках Fortran и C. Структура реализации методов генерации псевдослучайных чисел позволяет получить практически 100% эффективность параллельного вычислительного алгоритма. SPRNG поддерживает стандарты MPI и PVM и может реализовываться на различных платформах, в том числе PC-кластере.

7. ПРЕОБРАЗОВАНИЕ ФУРЬЕ. БЫСТРОЕ ПРЕОБРАЗОВАНИЕ ФУРЬЕ

Математические преобразования играют важную роль в численном анализе. Среди них наиболее известным является дискретное преобразование Фурье, так как существует быстрый алгоритм его вычисления – так называемый алгоритм быстрого преобразования Фурье (БПФ). В последние годы в связи с интенсивным развитием цифровой вычислительной техники внимание исследователей стали привлекать полные системы прямоугольных ортогональных функций Уолша, Хаара и др., для которых также существуют быстрые процедуры построения.

Ортогональные дискретные преобразования находят свое применение:

- в методах вычислений – при решении краевых задач для дифференциальных уравнений с частными производными, при интерполировании функций;
- при анализе временных рядов для определения спектра частот, вычисления автокорреляций;
- в области обработки изображений и речевых сигналов;
- при кодировании и декодировании информации и др.

7.1. Быстрое преобразование Фурье

Дискретное преобразование Фурье (ДПФ) вектора $\vec{x}$ длиной n определяется следующим образом:

$$y_m = \sum_{k=0}^{n-1} x_k \exp\left(i \frac{2\pi km}{n} \right),\ m = 0,1,...,n-1;\ \ i^2 = -1 . \qquad (7.1)$$

Обратное дискретное преобразование

$$x_k = \frac{1}{n} \sum_{m=0}^{n-1} y_m \exp\left(-i \frac{2\pi mk}{n} \right),\ k = 0,1,...,n-1 \qquad (7.2)$$

имеет подобное представление.

Обозначим $\omega_n = \exp\left(i\frac{2\pi}{n}\right) = \sqrt[n]{1}$ – главный корень n-й степени из единицы. Можно отметить следующие свойства введенного понятия: $\omega_n^{mn} = 1$, $(m = 0,1,2,...)$; $\omega_n^{mn/2} = -1$, $(m = 1,3,5,...)$; $\omega_n^{k+mn} = \omega_n^k$, $(k, m = 0,1,2,...)$ (см. рис. 7.1).

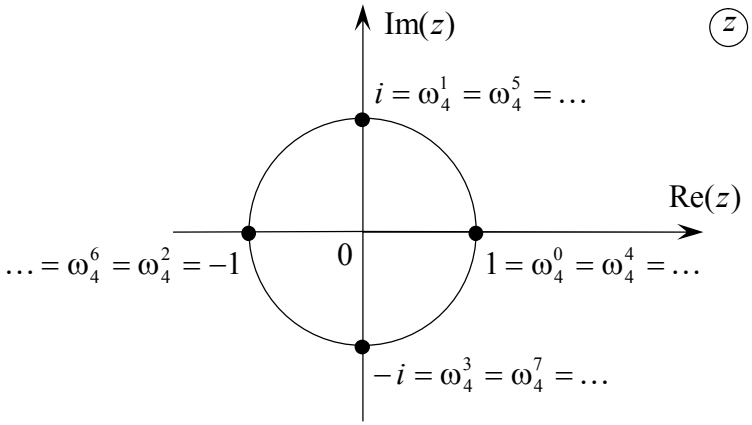


Рис. 7.1. Демонстрация некоторых свойств ω_n^k

Прямое вычисление ДПФ

$$y_m = \sum_{k=0}^{n-1} \omega_n^{km} x_k\ ,\ m = 0,1,...,n-1 \qquad (7.3)$$

потребует n комплексных умножений и n комплексных сложений на одну компоненту вектора $\vec{y}$ или $8n^2$ вещественных арифметических операций для вычисления вектора $\vec{y}$.

На протяжении последних двухсот лет математики неоднократно предлагали эффективные методы для вычисления ДПФ. В некоторых случаях получены результаты их успешного применения. Возможность рекурсивного представления (7.3) была известна Гауссу, который в 1805 году использовал этот подход для интерполирования траекторий астероидов Паллас и Джуно.

В 1942 году Даниэльсон и Ланцош представили вывод алгоритма эффективного вычисления ДПФ. Они показали, что (7.3) может быть записано как сумма двух ДПФ длиной $n/2$: одно формируется из компонентов вектора $\vec{x}$ с четными индексами $(x_0, x_2, x_4, ...)$, другое – из компонентов с нечетными индексами $(x_1, x_3, x_5, ...)$. Доказательство этого утверждения (лемма Даниэльсона – Ланцоша) следует из преобразований:

$$y_m = \sum_{k=0}^{n-1} \omega_n^{km} x_k = \sum_{k=0}^{n/2-1} \omega_n^{2km} x_{2k} + \sum_{k=0}^{n/2-1} \omega_n^{(2k+1)m} x_{2k+1} = \\ = \sum_{k=0}^{n/2-1} \omega_{n/2}^{km} x_{2k} + \omega_n^m \sum_{k=0}^{n/2-1} \omega_{n/2}^{km} x_{2k+1}; m = 0,1,...,n-1. \tag{7.4}$$

Замечательным в лемме Даниэльсона – Ланцоша (7.4) является то, что она может быть применена рекурсивно, если $n = 2^q, q \in \mathbb{N}$. Так, (7.3) можно представить в виде суммы четырех ДПФ длиной $n/4$:

$$y_m = \sum_{k=0}^{n/4-1} \omega_{n/4}^{km} x_{4k} + \omega_{n/2}^m \sum_{k=0}^{n/4-1} \omega_{n/4}^{km} x_{4k+2} + \\ + \omega_n^m \left(\sum_{k=0}^{n/4-1} \omega_{n/4}^{km} x_{4k+1} + \omega_{n/2}^m \sum_{k=0}^{n/4-1} \omega_{n/4}^{km} x_{4k+3} \right) = \\ = \sum_{s=0}^{4-1} \left[\omega_n^{sm} \sum_{k=0}^{n/4-1} \omega_{n/4}^{km} x_{4k+s} \right]; m = 0,1,...,n-1. \tag{7.5}$$

Из формулы (7.5) очевидным является следующее следствие леммы:

$$y_m = \sum_{s=0}^{r-1} \omega_n^{sm} \left(\sum_{k=0}^{n/r-1} \omega_n^{rkm} x_{rk+s} \right); \quad m = 0,1,...,n-1. \tag{7.6}$$

Для нахождения компонент вектора $\vec{y}$ по (7.6) необходимо $n(2n/r + 2r) = 2n(n/r + r)$ комплексных операций или $8n(n/r + r)$ вещественных арифметических операций. При $r \approx \sqrt{n}$ количество вычислений в (7.6) по сравнению с (7.3) сокращается приблизительно в $\sqrt{n}/2$ раз. Чем больше n, тем значительнее уменьшается число операций.

При $n=2^q$ очевидное уменьшение количества комплексных сложений и умножений получается, если (7.3), используя (7.6), представить в виде сумм:

$$
\begin{aligned}
y_m &= \sum_{s_0=0}^{1} \omega_n^{ms_0} \left(\sum_{k=0}^{n_1-1} \omega_{n_1}^{km} x_{2k+s_0} \right) = \sum_{s_0=0}^{1} \omega_n^{ms_0} \left(\sum_{k=0}^{n_1-1} \omega_{n_1}^{km} z_k^{(s_0)} \right) = \\
&= \sum_{s_0=0}^{1} \omega_n^{ms_0} \left(\sum_{s_1=0}^{1} \omega_{n_1}^{ms_1} \left(\sum_{k=0}^{n_2-1} \omega_{n_2}^{km} z_{2k+s_1}^{(s_0)} \right) \right) = \\
&= \sum_{s_0=0}^{1} \omega_n^{ms_0} \left(\sum_{s_1=0}^{1} \omega_{n_1}^{ms_1} \left(\sum_{k=0}^{n_2-1} \omega_{n_2}^{km} z_k^{(s_0,s_1)} \right) \right) = \\
&= \sum_{s_0=0}^{1} \omega_n^{ms_0} \left(\sum_{s_1=0}^{1} \omega_{n_1}^{ms_1} \left(\ldots \left(\sum_{s_{q-1}=0}^{1} \omega_{n_{q-1}}^{ms_{q-1}} \left(\sum_{k=0}^{n_q} \omega_{n_q}^{km} z_k^{(s,s,\ldots,s)} \right) \right) \right) \right)
\end{aligned}
\tag{7.7}
$$

$m=0,1,\ldots,n-1$.

Здесь $n_l = n/2^l; z_k^{(s_0,\ldots,s_{l-1})} = z_{2k+s_{l-1}}^{(s_0,\ldots,s_{l-2})}; k=0,\ldots,n_l-1; l=1,\ldots,q;$

$s_0, s_1, \ldots, s_{q-1} = 0,1; z_k^{(s_0)} = x_{2k+s_0}$.

В (7.7) для нахождения одной компоненты вектора $\vec{y}$ требуется $q=\log_2 n$ комплексных умножений и q комплексных сложений или $8n\log_2 n$ арифметических операций. Различие между $n\log_2 n$ и n^2 при $n \gg 1$ велико. Так, например, при $n=2^{10}=1024$ оно составляет $1024/10 \approx 100$ раз.

Существование алгоритма БПФ стало широко известным лишь в середине 60-х годов прошлого века из опубликованной статьи Кули и Тьюки.

На рис. 7.2 представлен последовательный алгоритм БПФ Кули – Тьюки, записанный на псевдокоде в итерационной форме. Эта программа использует два массива R и S длиной n для вычисления преобразования. Итерационный алгоритм содержит внешний цикл с $\log_2 n$ итерациями. Каждая итерация состоит из двух внутренних циклов. В первом производится обновление массива S соответствующими элементами массива R, рассчитанными на предыдущей

итерации. Второй внутренний цикл предназначен для вычисления значений R_i по соответствующей комбинации $S_{i_1} + \omega_n^{i_3} S_{i_2}$.

```
do i = 0, n-1          ! инициализация массива R
   R(i) = x(i)
end do
   m = log2(n)
do k = 0, m-1
  do i=0, n-1
    S(i) = R(i)
  end do
  do i = 0, n-1
     (b0, b1, ..., bm-1) = binary(i) ! определение двоичного
                                     ! представления i
     i1 = (b0, b1, ..., bk-1, 0, bk+1, ..., bm-1)  ! вычисление
                                                   ! индексов
     i2 = (b0, b1, ..., bk-1, 1, bk+1, ..., bm-1)
     i3 = (bk, bk-1, ..., b0, 0, 0, ..., 0)
    R(i) = S(i1) + ω**i3 * S(i2)
 end do
end do
do i = 0, n-1        ! передача результата массиву y
 y(i)=R(i)
end do
```

Рис. 7.2. Последовательный итерационный алгоритм БПФ

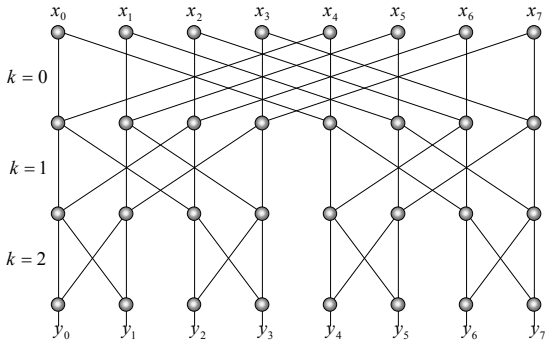


Рис. 7.3. Диаграмма вычисления БПФ для $n = 8$

На рис. 7.3 представлена диаграмма, иллюстрирующая шаги вычислений итерационного алгоритма БПФ (рис.7.2) для $n = 8$. Формулы, показывающие ход вычислительного процесса, могут быть получены на основе (7.5):

$$y_m = \left\{\left(x_0 + (-1)^m x_4\right) + \omega_8^{2m}\left(x_2 + (-1)^m x_6\right)\right\} +$$
$$+ \omega_8^m \left\{\left(x_1 + (-1)^m x_5\right) + \omega_8^{2m}\left(x_3 + (-1)^m x_7\right)\right\}; m = 0,...,7.$$

7.2. Параллельная реализация БПФ

Чтобы получить мелкозернистый параллельный алгоритм быстрого преобразования Фурье, распределим исходный вектор $\vec{x} \in \mathbb{R}^n$ равномерно между p процессорами, где $p = 2^u, u \in \mathbb{N}$ (рис. 7.4, $n = 8$, $p = 2$, $u = 1$). При его реализации на многопроцессорной вычислительной технике с распределенной памятью, как видно из рисунка, обмен данных между процессорами требуется только на первых $u = \log_2 p$ итерациях, после этого все вычисления будут выполняться параллельно.

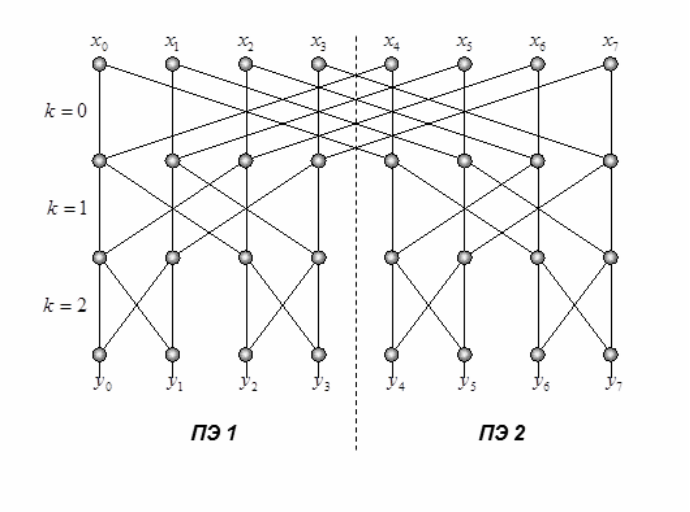


Рис. 7.4. Распределение исходных данных для БПФ между процессорами

Проведем теоретический анализ ускорения и эффективности построенного параллельного алгоритма БПФ. Время, затрачиваемое на осуществление вычислений, состоит из двух слагаемых. Первая составляющая T_p^{comp} есть эффективное время проведения арифметических операций (сложений и умножений) на p-процессорной вычислительной системе. Вторая составляющая T_p^{comm} представляет собой эффективное время, затрачиваемое на обмен данными между процессорами для организации рассмотренного распределенного мелкозернистого алгоритма БПФ. В рассматриваемом случае

$$T_p^{comp} = O\left(\frac{n}{p}\log_2 n\right); T_p^{comm} = O\left(p\log_2 p\right);$$

$$T_p = T_p^{comp} + T_p^{comm} = O\left(\frac{n}{p}\log_2 n + p\log_2 p\right).$$

Тогда теоретические оценки ускорения и эффективности параллельного алгоритма будут

$$S_p = \frac{T_1}{T_p} = O\left(\frac{p}{1+\frac{p^2\log_2 p}{n\log_2 n}}\right); \; E_p = \frac{S_p}{p} = O\left(\frac{1}{1+\frac{p^2\log_2 p}{n\log_2 n}}\right). \qquad (7.8)$$

Из этих формул следует, что при $n \gg p$ $S_p \approx p, E_p \approx 1$. Отметим, что при увеличении числа процессоров p эффективность алгоритма, при фиксированном n, падает за счет увеличения количества обменов относительно числа арифметических операций.

7.3. Алгоритм БПФ с использованием перестановок

Описанный выше параллельный алгоритм БПФ имеет две фазы. В первой – обмены данными между процессорами необходимы на каждой итерации алгоритма, а во второй нет коммуникационных операций. Рассмотрим другой параллельный алгоритм БПФ, в котором на этапе вычислений отсутствуют межпроцессорные обмены данными,

однако для подготовки вычислительных этапов требуется переупорядочивание данных с использованием пересылок.

Для понимания организации параллельных вычислений в процедуре БПФ с использованием перестановок первоначально рассмотрим последовательную версию этого алгоритма.

В формуле (7.6) положим $n = q^2$, $q > 0$ – целое. Тогда (7.6) примет вид

$$y_m = \sum_{s=0}^{q-1} \omega_n^{sm} \left(\sum_{k=0}^{q-1} \omega_n^{qkm} x_{qk+s} \right); \quad m = 0,1,...,n-1. \tag{7.9}$$

Введем обозначения:

$\phi_{k,s} = x_{qk+s}; k,s = 0,...,q-1;$ $y_m = y_{tq+l} = \psi_{t,l}; t,l = 0,...,q-1.$

В этом случае (7.9) можно записать следующим образом:

$$\psi_{t,l} = \sum_{s=0}^{q-1} \omega_n^{s(tq+l)} \left(\sum_{k=0}^{q-1} \omega_q^{k(tq+l)} \phi_{k,s} \right) = \sum_{s=0}^{q-1} \omega_n^{s(tq+l)} \underline{\underline{\left(\sum_{k=0}^{q-1} \omega_q^{kl} \phi_{k,s} \right)}}, \tag{7.10}$$

$t,l = 0,...,q-1.$

Подчеркнутые слагаемые в скобках есть ДПФ, примененное к векторам-столбцам матрицы $\Phi = \left\{ \phi_{k,s} \right\}$.

Обозначим

$$v_{l,s} = \sum_{k=0}^{q-1} \phi_{k,s} \omega_q^{lk}; l,s = 0,...,q-1, \tag{7.11}$$

$v_{l,s}$ есть компоненты матрицы $V = \begin{pmatrix} v_{0,0} & v_{0,1} & \cdots & v_{0,q-1} \\ v_{1,0} & v_{1,1} & \cdots & v_{1,q-1} \\ \cdots & \cdots & \cdots & \cdots \\ v_{q-1,0} & v_{q-1,1} & \cdots & v_{q-1,q-1} \end{pmatrix}$.

Тогда

$$\psi_{t,l} = \sum_{s=0}^{q-1} v_{l,s}\omega_n^{s(tq+l)} = \sum_{s=0}^{q-1}\left(v_{l,s}\omega_n^{sl}\right)\omega_q^{st} = \\ = \sum_{s=0}^{q-1}\xi_{l,s}\omega_q^{st}; t,l = 0,...,q-1. \tag{7.12}$$

Здесь

$$\xi_{l,s} = v_{l,s}\omega_n^{sl}; s,l = 0,...,q-1. \tag{7.13}$$

При проведении вычислений по формулам (7.10) и (7.11) можно применять алгоритм быстрого преобразования Фурье.

Запишем этапы алгоритма БПФ с использованием перестановок:

1. Исходный вектор $\vec{x}$ разместим в матрицу Φ следующим образом:

$$\Phi = \begin{pmatrix} x_0 & x_1 & \dots & x_{q-1} \\ x_q & x_{q+1} & \dots & x_{2q-1} \\ \dots & \dots & \dots & \dots \\ x_{(q-1)q} & x_{(q-1)q+1} & \dots & x_{q^2-1} \end{pmatrix}.$$

2. При вычислении компонентов матрицы V выполним БПФ по столбцам матрицы Φ по формуле (7.10). Рассчитаем элементы матрицы $\Xi = \left\{\xi_{l,s}\right\}$ по (7.13).

3. Применим БПФ по строкам матрицы Ξ и найдем матрицу Ψ, компонентами которой являются компоненты вектора – результата преобразования $\vec{y}$.

$$\Psi = \begin{pmatrix} y_0 & y_1 & \dots & y_{q-1} \\ y_q & y_{q+1} & \dots & y_{2q-1} \\ \dots & \dots & \dots & \dots \\ y_{(q-1)q} & y_{(q-1)q+1} & \dots & y_{q^2-1} \end{pmatrix}.$$

Как видно из описания последовательного алгоритма БПФ с использованием перестановок, параллельную его версию целесообразно строить, применяя принцип декомпозиции. При этом число активированных процессоров p должно удовлетворять усло-

вию $p \le q = \sqrt{n}$. Распределение исходного вектора $\vec{x}$ по процессорам производится таким образом, чтобы у каждого процессорного элемента было *q/p* столбцов матрицы Ф.

Параллельный алгоритм выполняется в следующем порядке:

1. Каждый процессорный элемент применяет алгоритм БПФ к имеющимся столбцам матрицы. Вычисления производятся параллельно без обмена данными между ПЭ.

2. Производится транспонирование полученной матрицы V (7.10) с использованием глобальных коммуникационных процедур так, чтобы у каждого ПЭ было *q/p* строк матрицы *V*.

3. Каждый ПЭ умножает элементы матрицы V на соответствующие множители и вновь параллельно применяет БПФ к имеющимся данным. На этом этапе также нет обмена данными между ПЭ.

4. Производится сборка вектора $\vec{y}$ на требуемом вычислительном узле.

Рассмотрим теоретические оценки ускорения и эффективности параллельного алгоритма БПФ с использованием перестановок. В данном алгоритме дважды используется БПФ, причем оба раза оно выполняется параллельно без коммуникационных затрат. Кроме того, по п. 3 параллельного алгоритма необходимо *n/p* комплексных умножений. Поэтому время на выполнение арифметических операций

$$T_p^{comp} = O\left(\frac{2q}{p} q \log_2 q + \frac{n}{p}\right) = O\left(\frac{n}{p}\log_2 n + \frac{n}{p}\right) = O\left(\frac{n}{p}\log_2 n\right).$$

Для осуществления транспонирования матрицы (п. 2) каждый процессорный элемент производит пересылку q^2/p^2 элементов матрицы на остальные ПЭ. Тогда

$$T_p^{comm} = O(n) \quad \text{и} \quad T_p = O\left(\frac{n}{p}\log_2 n + n\right).$$

Оценки ускорения и эффективности алгоритма

$$S_p = O\left(\frac{p}{1+\frac{p}{\log_2 n}}\right); \quad E_p = O\left(\frac{1}{1+\frac{p}{\log_2 n}}\right) \tag{7.14}$$

показывают, что при $n \gg p$ $S_p \approx p, E_p \approx 1$ и что алгоритм БПФ с использованием перестановок обладает большей производительностью, чем алгоритмы, использующие коммуникации типа *point-to-point (*см. п. 7.2*)*.

8. ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ РЕШЕНИЯ ЗАДАЧИ КОШИ ДЛЯ СИСТЕМЫ ОДУ

Моделирование поведения сложных динамических систем, протекания химических реакций, взаимодействия биологических видов приводит к необходимости решения систем обыкновенных дифференциальных уравнений (ОДУ) большой размерности. Сокращение времени решения и, следовательно, времени моделирования возможно при использовании мультипроцессорных систем, эффективность работы которых существенно зависит от характеристик применяемых параллельных алгоритмов.

8.1. Постановка задачи и обзор методов ее решения

Рассмотрим систему нелинейных обыкновенных дифференциальных уравнений вида

$$\begin{cases} \dfrac{dy_1(t)}{dt} = f_1(t, y_1(t), y_2(t), ..., y_n(t)); \\ \dfrac{dy_2(t)}{dt} = f_2(t, y_1(t), y_2(t), ..., y_n(t)); \\ \qquad\qquad ... \\ \dfrac{dy_n(t)}{dt} = f_n(t, y_1(t), y_2(t), ..., y_n(t)); \end{cases} \tag{8.1}$$

с начальными условиями:

$$y_1(t_0) = y_{10}; y_2(t_0) = y_{20}; ...; y_n(t_0) = y_{n0}.$$

Задачу (8.1) можно переписать в векторном виде:

$$\frac{d\vec{y}}{dt} = \vec{f}(t, \vec{y}); \ \vec{y}(t_0) = \vec{y}_0 = \begin{pmatrix} y_{10} \\ y_{20} \\ \cdots \\ y_{n0} \end{pmatrix}. \tag{8.2}$$

Здесь $\vec{y}(t)=\begin{pmatrix} y_1(t) \\ y_2(t) \\ \dots \\ y_n(t) \end{pmatrix}$ и $\vec{f}(t,\vec{y}(t))=\begin{pmatrix} f_1(t,y_1(t),...,y_n(t)) \\ f_2(t,y_1(t),...,y_n(t)) \\ \dots \\ f_n(t,y_1(t),...,y_n(t)) \end{pmatrix}$.

Для решения задачи Коши (8.1) используются аналитические и численные методы. Численное решение задачи Коши (8.1) рассчитывается для конечного набора значений независимой переменной t: $t_0,t_1,t_2,...,t_M$. Пусть точное решение задачи Коши (8.1) в точке t_m есть $\vec{y}(t_m)$, а приближенное обозначим через $\vec{y}_m$. Приближенное решение $\vec{y}_m$ вычисляется по найденным ранее одному или нескольким значениям $\vec{y}_{m-1},\vec{y}_{m-2},...,\vec{y}_1,\vec{y}_0$. Большинство численных методов решения задачи Коши для системы ОДУ можно представить в виде

$$\vec{y}_{m+1}=\vec{F}(\vec{y}_{m-q},\vec{y}_{m-q+1},...,\vec{y}_m,\vec{y}_{m+1},...,\vec{y}_{m+s},...),m=q,q+1,..., \quad (8.3)$$

где $\vec{F}$ – известная вектор-функция указанных аргументов и значений независимой переменной t, вид которой определяется выбранным способом построения численного метода; $q,s\in\mathbb{N}_0$. В настоящее время сложилась следующая классификация численных методов решения задачи Коши для системы ОДУ:

- при $q=0$ и $0\le s\le 1$ – одношаговые методы;
- при $q\ge 1$ или $s>1$ – многошаговые методы;
- при $s=0$ – явные методы;
- при $s=1$ – неявные методы;
- при $s>1$ – многошаговые формулы Коуэлла с забеганием вперед.

При построении параллельных алгоритмов для решения задачи Коши для системы ОДУ, следуя классификации Гира, рассмотрим два подхода: функциональную декомпозицию, или распараллеливание по времени (*parallelism across time*), и декомпозицию по данным, или распараллеливание по пространству (*parallelism across space*).

8.2. Метод последовательных приближений Пикара

Распараллеливание задачи Коши для системы ОДУ по времени, на первый взгляд, кажется невыполнимым, поскольку процесс поиска численного решения задачи представляется последовательным: по начальным условиям из (8.1) найти поведение объекта в будущем при $t > t_0$. Причем при численном решении на каждом интервале $[t_m, t_{m+1}]$ такой прогноз должен осуществляться по рекуррентным формулам, используя приближенное решение, полученное на предыдущем шаге (8.3). Тем не менее существуют методы, позволяющие всю область поиска решения задачи (8.2) $[t_0, t_M]$ разбить на подобласти $t_0 = \tau_0 < \tau_1 < ... < \tau_p = t_M$ (количество подобластей совпадает с числом используемых процессорных элементов p) и в каждой подобласти $[\tau_\mu, \tau_{\mu+1}]$ решать задачу Коши одновременно и независимо, используя заранее найденное приближенное решение задачи при $t = \tau_\mu$. Одним из таких алгоритмов, который можно распараллелить по времени, является алгоритм последовательных приближений Пикара.

Рассмотрим задачу Коши для одного ОДУ первого порядка (см. формулу (8.1), n=1)

$$y'(t) = f(t, y(t)), \;\; t_0 \le t \le t_M, \;\; y(t_0) = y_0. \qquad (8.4)$$

Интегрируя дифференциальное уравнение, заменим эту задачу эквивалентным ей интегральным уравнением

$$y(t) = y_0 + \int_{t_0}^{t} f(\tau, y(\tau)) d\tau.$$

Решая это интегральное уравнение методом последовательных приближений, получим итерационный процесс Пикара

$$y^{(k)}(t) = y_0 + \int_{t_0}^{t} f(\tau, y^{(k-1)}(\tau)) d\tau, y^{(0)}(t) \equiv y_0, \qquad (8.5)$$

здесь k=1,2,3,… – номер приближения. Заметим, что на каждой k-й итерации этого процесса интегрирование выполняется аналитически.

Итерационный процесс (8.5) сходится к точному решению задачи (8.4) в области $t_0 \le t \le t_M$, если функция $f(t,y)$ по второму аргументу удовлетворяет условию Липшица в рассматриваемой области.

Для системы (8.2) метод последовательных приближений Пикара запишется следующим образом:

$$\vec{y}^{(k)}(t) = \vec{y}_0 + \int_{t_0}^{t} \vec{f}(\tau, \vec{y}^{(k-1)}(\tau))d\tau, \;\; \vec{y}^{(0)}(t) \equiv \vec{y}_0 . \qquad (8.6)$$

Рассмотрим численную реализацию метода последовательных приближений (8.6), в котором интегрирование выполняется с помощью квадратурных формул. Пусть $\vec{y}_m^{(k)}$ – приближенное решение задачи Коши (8.2) в узле t_m на *k*-й итерации. На каждом интервале $[t_m, t_{m+1}]$ приближенное решение $\vec{y}_{m+1}^{(k)}$ будем рассчитывать по формуле (8.6), в которой для вычисления интеграла воспользуемся квадратурной формулой трапеций:

$$\vec{I}_m^{(k-1)} = \frac{(t_{m+1} - t_m)}{2}\left[\vec{f}\left(t_m, \vec{y}_m^{(k-1)}\right) + \vec{f}\left(t_{m+1}, \vec{y}_{m+1}^{(k-1)}\right)\right], \qquad (8.7)$$

$$\vec{y}_{m+1}^{(k)} = \vec{y}_m^{(k)} + \vec{I}_m^{(k-1)}; \;\; \vec{y}_m^{(0)} = \vec{y}_0; \; m = 0,1,2,...,M-1; \quad k = 1,2,...,K. \qquad (8.8)$$

Заметим, что более трудоемкие (массовый расчет значений правых частей $\vec{f}$) для вычислений формулы (8.7) могут применяться в расчетах на каждой итерации одновременно и независимо на всех интервалах. Вычисления по формуле (8.8) для каждого *k* проводятся последовательно*:* сначала вычисляется $\vec{y}_1^{(k)}$, затем $\vec{y}_2^{(k)}$, …, и, наконец, $\vec{y}_M^{(k)}$. Однако для вычисления $\left\{\vec{y}_m^{(k)}\right\}$ потребуется гораздо меньше времени, чем для расчета $\left\{\vec{I}_m^{(k-1)}\right\}$. Итерационный процесс прекращается при незначительном изменении приближенного решения, например при $\left\|\vec{y}_M^{(k+1)} - \vec{y}_M^{(k)}\right\| < \varepsilon$.

Для МВС с распределенной памятью рассмотрим параллельный алгоритм метода последовательных приближений, использующий подход распараллеливания по времени. Область поиска решения

$[t_0, t_M]$ разобьем на непересекающиеся подобласти $[\tau_\mu, \tau_{\mu+1}]$ $(\mu = 0,..., p-1)$, количество которых совпадает с числом используемых процессорных элементов p. На каждом таком интервале приближенное решение рассчитывается в M/p узлах (рис. 8.1).

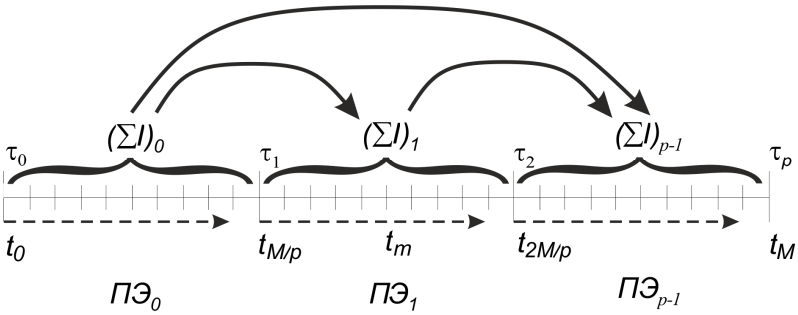


Рис. 8.1. Параллельный алгоритм решения задачи Коши методом Пикара. Сплошные линии отвечают передаче данных, шриховые – расчету приближенного решения в каждой подобласти поиска решения

На каждой k-й итерации μ-й ПЭ выполняет следующие действия (рис. 8.1):

- одновременно с другими ПЭ внутри своей подобласти $[\tau_\mu, \tau_{\mu+1}]$ вычисляет значения определенных интегралов $\vec{I}^{(k-1)}_{l+\mu M/p}, l = 0,..., M/p-1$ по формуле (8.7);

- выполняет суммирование полученных значений $\sum_{l=0}^{M/p-1} \vec{I}^{(k-1)}_{l+\mu M/p}$ и осуществляет пересылку рассчитанных сумм по межпроцессорной сети на ПЭ, номера которых больше μ;

- получив значения, производит вычисление $\vec{y}^{(k)}_{\mu M/p}$ и затем по формуле (8.8) находит значения приближенного решения в своей подобласти $[\tau_\mu, \tau_{\mu+1}]$:

$$\vec{y}^{(k)}_{l+1+\mu M/p} = \vec{y}^{(k)}_{l+\mu M/p} + \vec{I}^{(k-1)}_{l+\mu M/p};\ l = 0,1,2,...,M/p-1.$$

В представленном алгоритме вычисления интегралов и приближенного решения задачи Коши проводятся всеми ПЭ одновременно и независимо.

Заметим, что точность вычислений в методе Пикара существенно зависит от величины интервала интегрирования $[t_0, t_M]$ и может быть повышена за счет применения адаптивных квадратурных формул из главы 6.

Другой способ построения параллельных алгоритмов (*parallelism across space*) использует декомпозицию системы ОДУ (8.2) на подсистемы и одновременное решение полученных подсистем. Это может быть выполнено за счет массовых параллельных вычислений правых частей уравнений системы (8.2). В общем случае такой способ не обеспечивает баланс вычислительной нагрузки между процессорами в силу различного функционального представления правых частей ОДУ (8.2). Причем на каждом шаге по времени требуется проводить обмен данными для обеспечения последующих расчетов. Альтернативой является использование волновой релаксации, суть которой заключается в выполнении некоторого числа вычислительных шагов в каждой подсистеме, прежде чем осуществлять межпроцессорный обмен данными для продолжения вычислений. Основная проблема волновой релаксации заключается в сильной чувствительности получаемого приближенного решения от способа декомпозиции системы на подсистемы (желательно размещать взаимозависимые переменные в одной и той же подсистеме).

8.3. Параллельная реализация явного метода Рунге – Кутты

Метод Рунге – Кутты позволяет получать одношаговые разностные схемы различного порядка точности, в которых в зависимости от точности выбранной формулы правая часть системы может на рассматриваемом шаге вычисляться несколько раз.

Для построения вычислительных схем методов Рунге – Кутты четвертого порядка в тейлоровском разложении искомого решения $\vec{y}(t)$ учитываются члены, содержащие степени шага $h = t_{m+1} - t_m$ до четвертой включительно, наиболее часто используемой из которых является следующая:

$$\vec{y}_{m+1} = \vec{y}_m + (\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)/6, \qquad (8.9)$$

где

$$\vec{k}_1 = h\vec{f}(t_m, \vec{y}_m),$$
$$\vec{k}_2 = h\vec{f}(t_m + h/2, \vec{y}_m + \vec{k}_1/2),$$
$$\vec{k}_3 = h\vec{f}(t_m + h/2, \vec{y}_m + \vec{k}_2/2),$$
$$\vec{k}_4 = h\vec{f}(t_m + h, \vec{y}_m + \vec{k}_3).$$

Схема (8.9) на каждом шаге h требует четырех последовательных вычислений правой части системы ОДУ для определения значений $\vec{k}_1, \vec{k}_2, \vec{k}_3, \vec{k}_4$.

Поскольку в описанной выше вычислительной схеме наиболее трудоемкой является операция расчета правых частей ОДУ при вычислении $\vec{k}_i$ ($i = 1,2,3,4$), то основное внимание следует уделить распараллеливанию этой операции. Здесь будет применяться подход декомпозиции уравнений системы ОДУ на подсистемы. Поэтому для инициализации рассмотрим следующую схему декомпозиции данных по имеющимся процессорным элементам с локальной памятью: на каждый μ-й ПЭ ($\mu = 0,..., p-1$) распределяется *n/p* дифференциальных уравнений и вектор $\vec{y}_0 \in \mathbb{R}^n$. Далее расчеты производятся по следующей схеме:

1) на каждом ПЭ одновременно вычисляются *n/p* соответствующих компонент вектора $\vec{k}_1$ по формуле $\left[\vec{k}_1\right]_\mu = h\left[\vec{f}(t_m, \vec{y}_m)\right]_\mu$;

2) для обеспечения второго расчетного этапа необходимо провести сборку[†] вектора $\vec{k}_1$ целиком на каждом ПЭ. Затем независимо выполняется вычисление компонент вектора $\vec{k}_2$ по формуле $\left[\vec{k}_2\right]_\mu = h\left[\vec{f}(t_m + h/2, \vec{y}_m + \vec{k}_1/2)\right]_\mu$;

[†] Сборка вектора – операция межпроцессорной передачи данных, в результате выполнения которой каждым ПЭ принимаются компоненты вектора, рассчитываемые на других ПЭ, и объединяются в полноразмерный вектор .

3) проводится сборка вектора $\vec{k}_2$ на каждом ПЭ, вычисляются компоненты вектора $\vec{k}_3$: $\left[\vec{k}_3\right]_\mu = h\left[\vec{f}(t_m + h/2, \vec{y}_m + \vec{k}_2/2)\right]_\mu$;

4) проводится сборка вектора $\vec{k}_3$ на каждом ПЭ, вычисляются компоненты вектора $\vec{k}_4$: $\left[\vec{k}_4\right]_\mu = h\left[\vec{f}(t_m + h, \vec{y}_m + \vec{k}_3)\right]_\mu$;

5) рассчитываются с идеальным параллелизмом компоненты вектора $\vec{y}_{m+1}$: $\left[\vec{y}_{m+1}\right]_\mu = \left[\vec{y}_m\right]_\mu + (\left[\vec{k}_1\right]_\mu + 2\left[\vec{k}_2\right]_\mu + 2\left[\vec{k}_3\right]_\mu + \left[\vec{k}_4\right]_\mu)/6$

и производится сборка вектора $\vec{y}_{m+1}$ на каждом ПЭ. Если необходимо продолжить вычислительный процесс, то полагается $m = m+1$ и осуществляется переход на п. 1.

Заметим, что в данном алгоритме производится четыре операции вычисления вектора правых частей ОДУ, шестнадцать операций сложения векторов и умножения вектора на число и четыре операции глобальной сборки векторов.

В случае, когда $\vec{f}(t, \vec{y}(t)) \equiv A \times \vec{y}(t)$, где $A \in \mathbb{R}^{n \times n}$ – матрица с постоянными коэффициентами, можно получить алгоритм, который имеет меньшее число межпроцессорных пересылок данных на одной итерации метода Рунге – Кутты. Для этого построим оператор перехода для одного вычислительного шага, позволяющий определить значение вектора неизвестных на следующей итерации $\vec{y}_{m+1}$ через $\vec{y}_m$. Преобразуем значения $\vec{k}_1, \vec{k}_2, \vec{k}_3, \vec{k}_4$:

$$\vec{k}_1 = hA\vec{y}_m,$$

$$\vec{k}_2 = hA(\vec{y}_m + \vec{k}_1/2) = \vec{k}_1 + (h/2)A\vec{k}_1 = (E + (h/2)A)\vec{k}_1 =$$
$$= h(E + (h/2)A)A\vec{y}_m,$$

$$\vec{k}_3 = hA(\vec{y}_m + \vec{k}_2/2) = \vec{k}_1 + (h/2)A\vec{k}_2 = \vec{k}_1 + (h/2)A(\vec{k}_1 + (h/2)A\vec{k}_1) =$$
$$= h(E + (h/2)A + (h/2)^2 A^2)A\vec{y}_m,$$

$$\vec{k}_4 = hA(\vec{y}_m + \vec{k}_3) = \vec{k}_1 + (hA + (h^2/2)A^2 + (h^3/4)A^3)\vec{k}_1 =$$
$$= h(E + hA + (h^2/2)A^2 + (h^3/4)A^3)A\vec{y}_m,$$

и, подставив их в (8.9), получим

$$\vec{y}_{m+1} = B\vec{y}_m; B = E + hA(E + (h/2)A(E + (h/3)A(E + (h/4)A))), \quad (8.10)$$

где B – оператор перехода.

Рассчитав заранее матрицу B, получим вычислительный алгоритм, в котором каждое новое значение вектора $\vec{y}_{m+1}$ определяется за один шаг умножением матрицы B на вектор.

При параллельной реализации (8.10) умножение блочных строк матрицы B на вектор $\vec{y}_m$ будет проводиться одновременно и независимо так, как это описано в главе 2. Потребуется только после каждого вычислительного шага по (8.10) проводить сборку вектора $\vec{y}_{m+1}$ на всех процессорных элементах.

В заключение отметим, что, кроме явных методов Рунге – Кутты, используют и неявные методы Рунге – Кутты решения задачи Коши.

8.4. Параллельная реализация методов Адамса. Схема «предиктор – корректор»

При решении задачи Коши методами Рунге – Кутты необходимо вычислять правые части обыкновенных дифференциальных уравнений (8.2) в нескольких промежуточных точках на каждом шаге интегрирования. Количество таких вычислений зависит от порядка используемого метода. Заметим, что после того, как решение системы ОДУ определено в нескольких точках $t_0, t_1, ..., t_q$, можно применить алгоритмы интерполяции и сократить количество вычислений правых частей ОДУ (запоминая рассчитанные на предыдущих шагах значения) для получения вектора решения $\vec{y}_{q+1}$. Методы такого рода называют многошаговыми или многоточечными. Известно несколько типов таких методов. Алгоритмы многоточечных методов основываются на аппроксимации интерполяционными полиномами либо правых частей ОДУ, либо интегральных кривых.

Рассмотрим экстраполяционную явную формулу Адамса – Башфорта

$$\vec{y}_{m+1} = \vec{y}_m + (h/24)[55\vec{f}_m - 59\vec{f}_{m-1} + 37\vec{f}_{m-2} - 9\vec{f}_{m-3}], \quad (8.11)$$

где $\vec{f}_{m-i} = \vec{f}(t_m - ih, \vec{y}_{m-i}), i = 0,...,3;\ m = 3,4,...$.

Формула (8.11) имеет пятый порядок локальной погрешности и четвертый – глобальной.

Также приведем обладающую подобными параметрами погрешности неявную интерполяционную формулу Адамса – Моултона:

$$\vec{y}_{m+1} = \vec{y}_m + (h/24)[9\vec{f}_{m+1} + 19\vec{f}_m - 5\vec{f}_{m-1} + \vec{f}_{m-2})], \quad m = 2,3,\ldots . \tag{8.12}$$

Здесь искомая величина $\vec{y}_{m+1}$ используется для вычисления значения выражения $\vec{f}_{m+1} = \vec{f}(t_{m+1}, \vec{y}_{m+1})$, которое входит в правую часть. Формулу (8.12) можно рассматривать как систему нелинейных уравнений относительно неизвестных компонент вектора $\vec{y}_{m+1}$. Для решения такой системы наиболее часто используются метод простой итерации или метод Ньютона. Для сокращения вычислительных затрат решение, определенное по экстраполяционной формуле (8.11), обычно выбирается в качестве начального приближения для интерполяцинной формулы. Поэтому (8.11) рассматривается как формула прогноза, т.е.

$$\tilde{\vec{y}}_{m+1} = \vec{y}_m + (h/24)[55\vec{f}_m - 59\vec{f}_{m-1} + 37\vec{f}_{m-2} - 9\vec{f}_{m-3})], \tag{8.13}$$

а формула (8.12)

$$\vec{y}_{m+1} = \vec{y}_m + (h/24)[9\tilde{\vec{f}}_{m+1} + 19\vec{f}_m - 5\vec{f}_{m-1} + \vec{f}_{m-2})], \tag{8.14}$$

(здесь $\tilde{\vec{f}}_{m+1} = \vec{f}(t_{m+1}, \tilde{\vec{y}}_{m+1})$) является формулой коррекции, поскольку она имеет меньшую локальную погрешность вычислений по сравнению с (8.11). Последовательное применение (8.13) и (8.14) носит название схемы «предиктор – корректор». Недостатком методов Адамса является необходимость иметь перед началом вычислений решение в нескольких предыдущих узлах и усложение вычислительных схем для неравномерного шага интегрирования. Необходимые для корректного начала расчетов по формулам (8.13) и (8.14) стартовые значения $\vec{y}_1, \vec{y}_2, \vec{y}_3$ можно вычислить, например, по явному методу Рунге – Кутты (8.9).

Параллельная реализация методов Адамса, применяемых по схеме «предиктор – корректор», осуществляется в рамках подхода распараллеливания по пространству.

1. *Декомпозиция данных*.

Пусть μ-й процессорный элемент ($\mu = 0,..., p-1$) решает n/p уравнений системы ОДУ (8.1), где p – количество процессоров.

0 ПЭ	**1 ПЭ**	**…**	***p-1* ПЭ**
$\left[\vec{y}_m\right]_0, \left[\vec{y}_{m-1}\right]_0,$ $\left[\vec{y}_{m-2}\right]_0, \left[\vec{y}_{m-3}\right]_0$	$\left[\vec{y}_m\right]_1, \left[\vec{y}_{m-1}\right]_1,$ $\left[\vec{y}_{m-2}\right]_1, \left[\vec{y}_{m-3}\right]_1$		$\left[\vec{y}_m\right]_{p-1}, \left[\vec{y}_{m-1}\right]_{p-1},$ $\left[\vec{y}_{m-2}\right]_{p-1}, \left[\vec{y}_{m-3}\right]_{p-1}$

Здесь $\left[\vec{y}\right]_\mu \in \mathbb{R}^{n/p}$ – вектор, содержащий n/p компонент $\vec{y}$ $(1+\mu n/p, \ldots, (\mu+1)n/p)$.

2. *Параллельный алгоритм* (необходимые для начала вычислений по многошаговому методу Адамса значения векторов $\vec{y}_1, \vec{y}_2, \vec{y}_3$ рассчитываются параллельно методом Рунге – Кутты четвертого порядка):

а) инициализация ($m = 3$): на каждом μ-м ПЭ вычисляем

$$\left[\vec{V}^0\right]_\mu = \left[\vec{f}_{m-3}\right]_\mu, \left[\vec{V}^1\right]_\mu = \left[\vec{f}_{m-2}\right]_\mu, \left[\vec{V}^2\right]_\mu = \left[\vec{f}_{m-1}\right]_\mu, \left[\vec{V}^3\right]_\mu = \left[\vec{f}_m\right]_\mu;$$

б) на каждом μ-м ПЭ выполняем шаг «предиктор»

$$\left[\tilde{\vec{y}}_{m+1}\right]_\nu = \left[\vec{y}_m\right]_\mu + \frac{h}{24}\left\{55\left[\vec{V}^3\right]_\mu - 59\left[\vec{V}^2\right]_\mu + 37\left[\vec{V}^1\right]_\mu - 9\left[\vec{V}^0\right]_\mu\right\};$$

в) для выполнения шага коррекции осуществляется сборка на каждом ПЭ $\tilde{\vec{y}}_{m+1}$ для вычисления $\left[\tilde{\vec{f}}_{m+1}\right]_\mu$;

г) на каждом ПЭ выполняется шаг коррекции

$$\left[\vec{y}_{m+1}\right]_\mu = \left[\vec{y}_m\right]_\mu + \frac{h}{24}\left\{9\left[\tilde{\vec{f}}_{m+1}\right]_\mu + 19\left[\vec{V}^3\right]_\mu - 5\left[\vec{V}^2\right]_\mu + \left[\vec{V}^1\right]_\mu\right\};$$

д) для оценки сходимости этапа коррекции производится вычисление нормы $\left\|\vec{y}_{m+1} - \tilde{\vec{y}}_{m+1}\right\|$ с помощью каждого ПЭ:

- если $\left\|\vec{y}_{m+1} - \tilde{\vec{y}}_{m+1}\right\| > \varepsilon$, то $\tilde{\vec{y}}_{m+1} = \vec{y}_{m+1}$ и переход на п. «в»,

- если $\left\|\vec{y}_{m+1}-\tilde{\vec{y}}_{m+1}\right\|<\varepsilon$, то $\left[\vec{V}^0\right]_\mu=\left[\vec{V}^1\right]_\mu$, $\left[\vec{V}^1\right]_\mu=\left[\vec{V}^2\right]_\mu$, $\left[\vec{V}^2\right]_\mu=\left[\vec{V}^3\right]_\mu$, $\left[\vec{V}^3\right]_\mu=\left[\vec{f}_{m+1}\right]_\mu$, присваиваем $m=m+1$ и возвращаемся на п. «б».

Получим оценку ускорения рассмотренного параллельного алгоритма. Оценку будем производить по соотношению временных затрат на выполнение одного шага коррекции. Для последовательной версии $T_1\approx R(n)\times t_a$, здесь $R(n)$ – количество арифметических операций для расчета правых частей системы ОДУ и вычислений по формулам коррекции, t_a – обобщенное время выполнения одной арифметической операции. Для параллельной версии имеем $T_p\approx\frac{R(n)}{p}t_a+(p-1)\frac{n}{p}t_{comm}$. Здесь t_{comm} – время на пересылку одного числа. Тогда ускорение можно оценить как

$$S_p=\frac{T_1}{T_p}\approx\frac{R(n)t_a}{\frac{R(n)t_a}{p}+\frac{(p-1)nt_{comm}}{p}}=\frac{p}{1+\frac{(p-1)n\kappa}{R(n)}},\ \text{где}\ \kappa=t_{comm}/t_a.$$

Из полученных формул следует, что чем выше вычислительная сложность расчета правых частей заданной системы ОДУ, тем большего ускорения можно достичь при использовании параллельного алгоритма «предиктор – корректор».

8.5. Неявные методы Рунге – Кутты и Гира для численного решения задачи Коши

Для решения жестких систем ОДУ наибольшее преимущество с точки зрения обеспечения получения устойчивого решения имеют неявные методы Рунге – Кутты и методы Гира.

Систему ОДУ вида (8.2), описывающую протекающие с различной скоростью процессы, будем называть жесткой на интервале $t_0<t<t_M$, если выполняются следующие условия на собственные

значения $\{\lambda_k(t), k=1,...,n\}$ матрицы Якоби $J(t,\vec{y}(t)) = \vec{f}_{\vec{y}}(t,\vec{y}) = \left[\frac{\partial f_i(t,\vec{y})}{\partial y_j}\right]_1^n$:

- $\mathrm{Re}\ \lambda_k(t) < 0, (k=1,2,...,n)$ для всех $t \in (t_0, t_M)$;

- число жесткости системы $S = \sup\limits_{t\in(t_0,t_M)} \left\{ \frac{\max\limits_{1\le k\le n} |\mathrm{Re}\ \lambda_k(t)|}{\min\limits_{1\le k\le n} |\mathrm{Re}\ \lambda_k(t)|} \right\} \gg 1$.

Неявные одношаговые формулы Рунге – Кутты записываются в виде:

$$\vec{y}_{m+1} = \vec{y}_m + \sum_{i=1}^{q} A_i \vec{k}_i; \quad m = 0,1,...,M-1; \tag{8.15}$$

$$\vec{k}_i = h\vec{f}(t_m + \alpha_i h, \vec{y}_m + \sum_{j=1}^{q} \beta_{ij} \vec{k}_j), i = 1,2,...,q.$$

Здесь $\{A_i\}, \{\alpha_i\}, \{\beta_{ij}\}$ – коэффициенты; q определяет вид формул (8.15), например, при $q=2$ получается двучленная формула Рунге – Кутты четвертого порядка:

$$\vec{y}_{m+1} = \vec{y}_m + (\vec{k}_1 + \vec{k}_2)/2,$$

$$\vec{k}_1 = h\vec{f}\left(t_m + \frac{3-\sqrt{3}}{6}h, \vec{y}_m + \frac{1}{4}\vec{k}_1 + \left(\frac{1}{4} - \frac{\sqrt{3}}{6}\right)\vec{k}_2\right),$$

$$\vec{k}_2 = h\vec{f}\left(t_m + \frac{3+\sqrt{3}}{6}h, \vec{y}_m + \left(\frac{1}{4} + \frac{\sqrt{3}}{6}\right)\vec{k}_1 + \frac{1}{4}\vec{k}_2\right).$$

Неявные многошаговые методы Гира имеют следующее представление:

$$\sum_{i=0}^{q} A_i \vec{y}_{m+1-i} = h\vec{f}(t_{m+1}, \vec{y}_{m+1}), \quad m = q-1, q, ..., M-1. \tag{8.16}$$

Например, формула метода Гира четвертого порядка ($q=4$ в (8.16)) имеет вид

$$\frac{25\vec{y}_{m+1}-48\vec{y}_m+36\vec{y}_{m-1}-16\vec{y}_{m-2}+3\vec{y}_{m-3}}{12}=h\vec{f}(t_{m+1},\vec{y}_{m+1}).$$

Для решения на каждом шаге $h=t_{m+1}-t_m$ систем нелинейных уравнений – для вычисления $\{\vec{k}_i\}$ в неявных одношаговых методах Рунге – Кутты и для расчета $\vec{y}_{m+1}$ в чисто неявных методах Гира – используется итерационный метод Ньютона. Заметим, что в методах Рунге – Кутты на каждом шаге приходится решать qn нелинейных уравнений, а в методах Гира лишь n.

Метод Ньютона для решения систем нелинейных уравнений $\vec{F}(\vec{x})=0$,

где $\vec{x}=\begin{pmatrix}x_1\\x_2\\\vdots\\x_n\end{pmatrix}$ – вектор неизвестных, $\vec{F}(\vec{x})=\begin{pmatrix}g_1(x_1,...,x_n)\\g_2(x_1,...,x_n)\\\vdots\\g_n(x_1,...,x_n)\end{pmatrix}$,

представляется следующим итерационным процессом:

$$J(\vec{x}_k)(\vec{x}_{k+1}-\vec{x}_k)=-\vec{F}(\vec{x}_k),\ \ k=0,1,2,\ldots\ , \qquad (8.17)$$

где $J(\vec{x})=\begin{pmatrix}\frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \ldots & \frac{\partial g_1}{\partial x_n}\\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \ldots & \frac{\partial g_2}{\partial x_n}\\ \ldots & \ldots & \ldots & \ldots\\ \frac{\partial g_n}{\partial x_1} & \frac{\partial g_n}{\partial x_2} & \ldots & \frac{\partial g_n}{\partial x_n}\end{pmatrix}$.

Таким образом, численная реализация неявных методов Рунге – Кутты и Гира заключается в задании достаточно точных начальных приближений и выполнении на каждом шаге h итерационного процесса метода Ньютона (8.17), требующего в общем случае решения

систем линейных алгебраических уравнений на каждой итерации. Прямые и итерационные методы решения линейных систем и способы их распараллеливания подробно были рассмотрены в главах 3 и 4.

В заключение отметим, что задачу Коши для уравнения n-го порядка можно свести к задаче Коши для системы уравнений первого порядка (8.2), которую можно решить по описанным выше параллельным алгоритмам.

8.6. Параллельный алгоритм для сплайновой системы обыкновенных дифференциальных уравнений

Используя интерполяционные кубические сплайны, можно для краевых задач получать сплайновые системы ОДУ, применяя гибкую систему независимых аппроксимаций.

Рассмотрим краевую задачу

$$\frac{\partial u}{\partial t} = a^2 \frac{\partial^2 u}{\partial x^2} + f(u,x,t), \quad \{0 \le x \le 1, 0 < t \le T\};$$

$$t = 0 : u(x,0) = \phi(x); \quad x = 0 : \lambda \frac{\partial u}{\partial x} = \alpha(u - U); \quad x = 1 : \lambda \frac{\partial u}{\partial x} = \alpha(U - u);$$

где $a^2, \alpha, T, U, \lambda$ – заданные константы. Сплайновая система обыкновенных дифференциальных уравнений итерационно-интерполяционного метода (ИИМ) первого приближения для такой задачи имеет вид:

$$\begin{aligned}
&\delta\left(2\dot{V}_0 + \dot{V}_1\right) = \frac{\lambda\left(V_1 - V_0\right)}{h} - \alpha\left(V_0 - U\right) + \delta\mu\left(2f_0 + f_1\right);\\
&\dot{V}_{i-1} + 4\dot{V}_i + \dot{V}_{i+1} = 6a^2 \Lambda V_i + \mu\left(f_{i-1} + 4f_i + f_{i+1}\right), \; i = \overline{1, n-1};\\
&\delta\left(2\dot{V}_n + \dot{V}_{n-1}\right) = \frac{\lambda\left(V_{n-1} - V_n\right)}{h} + \alpha\left(U - V_n\right) + \delta\mu\left(2f_n + f_{n-1}\right);\\
&\Lambda V_i = \left(V_{i-1} - 2V_i + V_{i+1}\right)/h^2;\\
&V_i(0) = \varphi_i, \; i = \overline{0, n}, \; \alpha \ge 0, \; \delta = \frac{\lambda h}{6a^2} > 0, \; 0 < t \le T.
\end{aligned} \qquad (8.18)$$

Эту систему можно переписать в нормальной векторной форме

$$\frac{d\vec{u}}{dt} = C\vec{u} + \mu\vec{f} + A^{-1}\vec{b}\ ,\ \vec{u}(0) = \vec{\phi};\qquad 0 < t \le T;\ \ C = A^{-1}B, \qquad (8.19)$$

где A, B – трехдиагональные квадратные матрицы размера $(n+1)(n+1)$:

$$A = \begin{pmatrix} 2 & 1 & 0 & \dots & 0 \\ 1 & 4 & 1 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 1 & 4 & 1 \\ 0 & \dots & 0 & 1 & 2 \end{pmatrix},\ B = \begin{pmatrix} -\nu_1 & \nu_2 & 0 & \dots & 0 \\ \gamma & -2\gamma & \gamma & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \gamma & -2\gamma & \gamma \\ 0 & \dots & 0 & \nu_2 & -\nu_3 \end{pmatrix},$$

$$\vec{b} = \begin{pmatrix} \alpha U/\delta \\ 0 \\ \vdots \\ 0 \\ \alpha U/\delta \end{pmatrix}, \vec{f} = \begin{pmatrix} f_0 \\ \vdots \\ \vdots \\ \vdots \\ f_n \end{pmatrix}, \vec{V} = \begin{pmatrix} V_0 \\ \vdots \\ \vdots \\ \vdots \\ V_n \end{pmatrix}, \vec{\phi} = \begin{pmatrix} \phi_0 \\ \vdots \\ \vdots \\ \vdots \\ \phi_n \end{pmatrix},$$

$$\nu_1 = (\frac{\lambda}{h} + \alpha)/\delta,\ \ \nu_2 = \lambda(h\delta),$$

$$\nu_3 = (\frac{\lambda}{h} + \alpha)/\delta,\ \ \gamma = 6\,a^2/h^2.$$

Численное решение системы ОДУ ИИМ (8.18) можно найти методом Рунге – Кутты с итерациями. Решение системы (8.19) определяется без итераций, но требует вычисления элементов матрицы C.

Так как трехдиагональная матрица A со строгим диагональным преобладанием, то элементы ее обратной матрицы убывают с удалением от главной диагонали.

Для определения элементов обратной матрицы A^{-1} используем идею метода Гаусса. Введем обозначения:

$$A=\begin{pmatrix} a_0 & b_0 & 0 & \cdots & 0 \\ c_1 & a_1 & b_1 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & c_n & a_n \end{pmatrix},\ D=A^{-1}=\begin{pmatrix} d_{0,0} & \cdots & d_{0,n} \\ d_{1,0} & \cdots & d_{1,n} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ d_{n,0} & \cdots & d_{nn} \end{pmatrix},$$

$$a_i-(b_i+c_i)=\Delta_i>0,\quad i=\overline{0,n}.$$

Так как $A^{-1}A = E$, то элементы m-го столбца обратной матрицы являются решением системы линейных уравнений с трехдиагональной матрицей:

$$\begin{aligned} &a_0\,d_{0,m}+b_0\,d_{1,m}=0;\\ &c_k\,d_{k-1,m}+a_k\,d_{k,m}+b_k\,d_{k+1,m}=0,\ k=\overline{1,m-1};\\ &c_m\,d_{m-1,m}+a_m\,d_{m,m}+b_m\,d_{m+1,m}=1;\\ &c_k\,d_{k-1,m}+a_k\,d_{k,m}+b_k\,d_{k+1,m}=0,\quad k=\overline{m+1,n-1};\\ &c_n\,d_{n-1,m}+a_n\,d_{n,m}=0,\ m=\overline{0,n}. \end{aligned} \tag{8.20}$$

Учитывая соотношения метода встречных прогонок, для системы (8.20) получим:

$$d_{i,m}=\eta_i\,d_{i+1,m}+\psi_i,\ i=\overline{0,n-1}; \tag{8.21}$$

$$d_{i,m}=\overline{\eta}_i\,d_{i-1,m}+\overline{\psi}_i,\quad i=n,\ n-1,...,1; \tag{8.22}$$

$$\eta_0=-\frac{b_0}{a_0},\eta_i=\frac{-b_i}{a_i+c_i\,\eta_i},\quad \psi_i=0,\ i=\overline{0,m-1};$$

$$\overline{\eta}_n=-\frac{c_n}{a_n},\ \eta_i=\frac{-c_i}{a_i+b_i\,\overline{\eta}_{i+1}},\ \overline{\psi}_i=0,\ i=n,n-1,\ldots,m+1.$$

Исключив из m-го уравнения системы (8.20) $d_{m-1,m}$ и $d_{m+1,m}$ с помощью формул (8.21) и (8.22), определим диагональный элемент

$$d_{i,m}=(a_m+c_m\eta_{m-1}+b_m\overline{\eta}_{m+1})^{-1}, \tag{8.23}$$

а затем и все остальные элементы m-го столбца матрицы A^{-1}.

Для упрощения правой части системы (8.19), учитывая локальные свойства кубических сплайнов, вычислим точно только трехдиагональную часть матрицы A^{-1}. Тогда в каждой системе уравнений (8.20), после определения диагонального элемента по формуле (8.23), вычисляются только значения $d_{i-1,m}$, $d_{i+1,m}$, а остальные элементы считаются нулевыми. Обозначив такую матрицу через $A_3^{-1} = \left[d_{i,j} \right]_0^n$, получим приближенное представление системы (8.19) в нормальной форме

$$\dot{\vec{V}} = C_5 \vec{V} + \vec{f} + A_3^{-1} \vec{b},\ \vec{V}(0) = \vec{\varphi},\ C_5 = A_3^{-1} B. \qquad (8.24)$$

Так как матрицы A_3^{-1} *и* B трехдиагональные, то элементы пятидиагональной матрицы C_5 можно вычислить по описанным в главе 2 параллельным алгоритмам матричного умножения.

Если C_5 – симметричная матрица, то достаточно вычислить только верхнюю треугольную часть этой матрицы.

Решение системы ОДУ (8.24) можно осуществить методом Рунге – Кутты на многопроцессорной вычислительной системе с распределенной памятью. В этом случае необходимая точность по пространственной переменной обеспечивается выбором достаточно большого числа промежутков n, а заданную точность по времени можно обеспечить применением расчетных формул соответствующей точности. При этом для любого $n \geq 5$ число обменов между соседними процессорами при вычислении правых частей системы (8.24) на каждом шаге по времени t не превосходит двух.

Для пошагового контроля точности решения задачи Коши можно использовать вложенные явные одношаговые формулы Рунге – Кутты. Например, для линейной системы ОДУ

$$\dot{\vec{V}} = C\vec{V}\ ,\ \vec{V}(0) = \vec{\phi},\ t \in (0,T],\ C = \left[c_{i,j} \right]_0^n.$$

Мерсоном была предложена схема

$$\vec{y}_{m+1} = \vec{y}_m + \left(\vec{k}_1 - 3\vec{k}_3 + 4\vec{k}_4 \right) / 2\ , m = \overline{0, l-1}, \qquad (8.25)$$

$$\vec{v}_{m+1} = \vec{y}_m + \left(\vec{k}_1 + 4\vec{k}_4 + \vec{k}_5 \right) / 6\ , m = \overline{0, l-1}, \qquad (8.26)$$

где h – шаг по времени, $t_m=mh$, $T=lh$.

$$\vec{k}_1 = hC\vec{y}_m,\ \vec{k}_2 = hC\left(\vec{y}_m + \vec{k}_1/3\right), \vec{k}_3 = hC\left(\vec{y}_m + \vec{k}_1/6 + \vec{k}_2/6\right),$$

$$\vec{k}_4 = hC\left(\vec{y}_m + \vec{k}_1/8 + 3\vec{k}_2/8\right), \vec{k}_5 = hC\left(\vec{y}_m + \vec{k}_1/2 - 3\vec{k}_3/2 + 2\vec{k}_4\right).$$

Расчетная формула (8.25) имеет четвертый порядок точности, а формула (8.26) на порядок точнее.

Запишем (8.25), (8.26) в операторной форме (аналогично п. 8.3)

$$\vec{y}_{m+1} = B_1\vec{y}_m,\ \vec{v}_{m+1} = B_2\vec{y}_m,\ ,\qquad (8.27)$$

где операторы перехода

$$B_1 = E + hC + h^2C^2/2 + h^3C^3/6,\ B_2 = B_1 + h^4C^4/24.$$

Операторная запись (8.27) позволяет уменьшить время получения решения на каждом шаге по t за счет предварительного параллельного вычисления матричных операторов.

Для контроля точности $\varepsilon>0$ на каждом шаге вычисляют

$$\delta^{m+1} = h^4\left\|C^4\vec{y}_m\right\|_c/24.$$

Если $\varepsilon > \delta^{m+1}$, то вновь осуществляется счет по формулам (8.27) с шагом $h/2$. Если $\varepsilon < \delta^{m+1}$, то производится проверка на возможность удвоения шага для вычисления $\vec{y}_{m+2}$.

9. РЕШЕНИЕ КРАЕВЫХ ЗАДАЧ ДЛЯ УРАВНЕНИЙ В ЧАСТНЫХ ПРОИЗВОДНЫХ МЕТОДОМ КОНЕЧНЫХ РАЗНОСТЕЙ

Дифференциальные уравнения в частных производных широко используются при математическом моделировании процессов и явлений в современных технических устройствах и технологических аппаратах, при исследовании окружающего нас мира, при планировании экономики и финансов, изучении путей развития общества. Аналитическое решение таких уравнений можно получить только для некоторых простых случаев и канонических областей. Поэтому для решения многих математических задач, содержащих уравнения в частных производных, часто привлекаются приближенные численные методы: методы конечных разностей, методы конечных элементов или спектральные методы.

Основная идея таких методов заключается в дискретизации области поиска решения и дифференциальной постановки задачи. В результате такой дискретизации получается система линейных или нелинейных алгебраических уравнений, решение которой и есть приближенное решение исходной дифференциальной задачи. Именно процесс получения решения зачастую является главным источником параллелизма методов решения краевых задач для уравнений в частных производных.

Рассмотрим линейное дифференциальное уравнение второго порядка от функции двух переменных, записанное в следующем виде:

$$a(x,y)\frac{\partial^2 u}{\partial x^2} + 2b(x,y)\frac{\partial^2 u}{\partial x \partial y} + c(x,y)\frac{\partial^2 u}{\partial y^2} + d(x,y)\frac{\partial u}{\partial x} + \\ + e(x,y)\frac{\partial u}{\partial y} + g(x,y)u = f(x,y); \qquad (x,y) \in D. \tag{9.1}$$

Здесь $a(x,y), b(x,y), c(x,y), d(x,y), g(x,y), f(x,y)$ – функции, значения которых определены внутри области D.

Если в области D $b^2(x,y) - a(x,y) \cdot c(x,y) < 0$, то говорят об уравнении (9.1) как об уравнении в частных производных эллиптического

типа, при $b^2(x,y) - a(x,y)\cdot c(x,y) = 0$ – параболического и при $b^2(x,y) - a(x,y)\cdot c(x,y) > 0$ – гиперболического типа.

Уравнение Пуассона $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = f(x,y)$ – это пример уравнения эллиптического типа, которое применяется при моделировании стационарных процессов тепло- и электропроводности в телах, потенциальных течений несжимаемой жидкости.

Уравнение теплопроводности $\dfrac{\partial u}{\partial t} = a^2 \dfrac{\partial^2 u}{\partial x^2}$ есть пример уравнения параболического типа. Оно используется для математического моделирования нестационарной теплопроводности в твердых телах, для описания диффузионных процессов в жидкостях и газах.

Волновое уравнение $\dfrac{\partial^2 u}{\partial t^2} = b^2 \dfrac{\partial^2 u}{\partial x^2}$ относится к уравнениям второго порядка в частных производных гиперболического типа. Применяется при моделировании распространения волн или вибрации струн или мембран.

Для того, чтобы уравнение (9.1) имело единственное решение, необходимо к нему добавить дополнительные условия о поведении решения на границе области. Обычно для этого выбирают условия:

- первого рода, когда на границе задано значение решения

$$u\big|_{\Gamma} = \gamma(x,y); \qquad (x,y) \in \Gamma; \tag{9.2}$$

- второго рода, когда на границе задано значение производной от $u(x,y)$ по направлению внешней нормали

$$\left.\frac{\partial u}{\partial n}\right|_{\Gamma} = \gamma(x,y), \qquad (x,y) \in \Gamma; \tag{9.3}$$

- третьего рода, когда на границе задается комбинация значения искомой функции и ее производной

$$\alpha(x,y)\frac{\partial u}{\partial n} + \beta(x,y)\cdot u = \gamma(x,y), \qquad (x,y) \in \Gamma. \tag{9.4}$$

Рассмотрим разностные методы решения уравнений в частных производных эллиптического и параболического типов.

9.1. О решении задачи Дирихле для уравнения Пуассона в прямоугольнике с помощью метода конечных разностей

Пусть необходимо найти решение уравнения эллиптического типа с постоянными коэффициентами внутри прямоугольника, на границе которого известно значение решения:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f(x,y); \qquad (0<x<L_x, 0<y<L_y); \tag{9.5}$$

$$x=0: u=u_L(y), \quad 0\le y\le L_y;$$

$$x=L_x: u=u_R(y), \quad 0\le y\le L_y;$$

$$y=0: u=u_B(x), \quad 0\le x\le L_x;$$

$$y=L_y: u=u_T(x), \quad 0\le x\le L_x.$$

Функция $f(x,y)$ в (9.5) определена внутри прямоугольника.

В соответствии с методом конечных разностей сначала необходимо провести дискретизацию области поиска приближенного решения, т.е. перейти от области непрерывного изменения независимых переменных к сетке, представляющей собой конечное множество точек области, в которых будет находиться приближенное решение задачи.

Определим равномерную разностную сетку следующим образом:

$$\overline{\Omega}_h=\left\{(x_i,y_j),\ \begin{matrix} x_i=i\cdot h_x, i=0,\ldots,N_x+1; h_x=L_x/(N_x+1) \\ y_j=j\cdot h_y, j=0,\ldots,N_y+1; h_y=L_y/(N_y+1)\end{matrix}\right\}. \tag{9.6}$$

Здесь h_x, h_y – шаги сетки (расстояния между сеточными линиями) вдоль соответствующих направлений осей координат; N_x+1, N_y+1 – количество разбиений сторон прямоугольника.

Например, на рис. 9.1 представлена сетка $\overline{\Omega}_h$ (9.6) для $N_x=N_y=3$. Заметим, что в силу условий задачи решение в граничных узлах (множество граничных узлов обозначим γ_h, на рис. 9.1

они отмечены темными крестиками) известно и требуется оценить решение во внутренних узлах сетки $\Omega_h = \overline{\Omega}_h \setminus \gamma_h$ (светлые значки на рис. 9.1). Приближенное решение задачи в узлах сетки обозначим $\{v_{i,j}\}$, причем $v_{i,j} \approx u(x_i, y_j), i = 1, \ldots, N_x, j = 1, \ldots, N_y$.

На границе $v_{i,j} = u(x_i, y_j), (x_i, y_j) \in \gamma_h$.

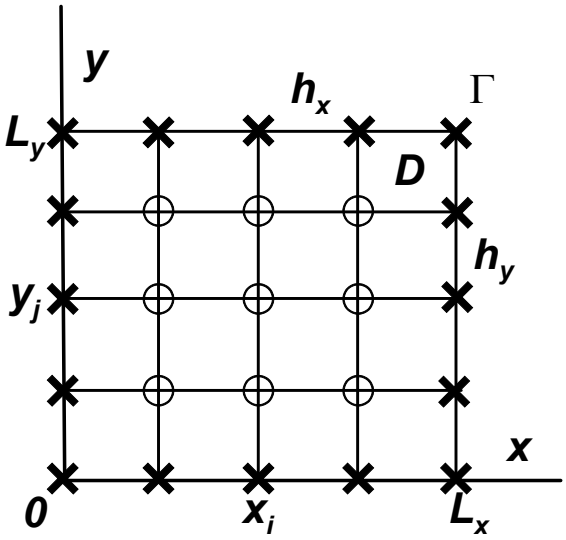


Рис. 9.1. Область $\overline{D} = D \bigcup \Gamma$ с нанесенной на нее сеткой $\overline{\Omega}_h$ для $N_x = N_y = 3$

Следующим шагом в методе конечных разностей является переход от дифференциальной постановки задачи для $u(x, y)$ (9.5) к конечно-разностной схеме, связывающей значения сеточной функции $v_{i,j}$ в узлах сетки в соответствии с выбранным сеточным шаблоном. Под сеточным шаблоном в методе конечных разностей понимается совокупность узловых точек сетки, в которых будут использоваться значения сеточной функции для аппроксимации производных в дифференциальной постановке (9.5). Для дискретизации уравнения задачи (9.5) будем использовать пятиточечный шаблон «крест» (рис. 9.2), который позволяет получить разностную схему для (9.5) со вторым порядком аппроксимации.

Для построения разностной схемы рассмотрим уравнение (9.5) во внутреннем узле сетки (x_i, y_j) и заменим вторые производные в узлах сетки по формулам численного дифференцирования:

$$\left(\frac{\partial^2 u}{\partial x^2}\right)\Big|_{(x_i,y_j)} \approx \frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{h_x^2}; \left(\frac{\partial^2 u}{\partial y^2}\right)\Big|_{(x_i,y_j)} \approx \frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{h_y^2}. \quad (9.7)$$

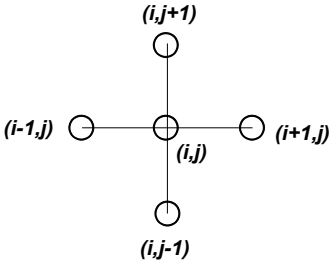


Рис. 9.2. Разностный шаблон «крест»

Подставим эти приближенные представления вторых производных во внутреннем узле (x_i, y_j) в уравнение (9.5). Тогда получим разностную схему

$$\frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{h_x^2} + \frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{h_y^2} = f(x_i, y_j); (x_i, y_j) \in \Omega_h; \quad (9.8)$$

$$(i = 1, \ldots N_x; j = 1, \ldots, N_y).$$

Добавляя к (9.8) известные на границе значения сеточной функции, получаем систему из $N_x \times N_y$ линейных алгебраических уравнений с $N_x \times N_y$ неизвестными $\{v_{i,j}\}$. Матрица этой системы содержит только пять ненулевых диагоналей и имеет диагональное преобладание ($|a_{i,i}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{N_x N_y} |a_{i,j}|, i = 1, \ldots, N_x N_y$). В (9.9) представлена система уравнений (9.8) в матрично-векторном виде при $N = N_x = N_y = 3$. Здесь $\{f_{i,j}^*\}$ – модифицированные умножением на h^2 правые части (9.8). В общем случае матрица разностной схемы имеет блочный вид, разме-

ры блоков $N \times N$, на главной диагонали расположены блоки – трехдиагональные матрицы, на соседних верхней и нижней диагоналях – блоки – диагональные матрицы. Остальные блоки состоят из нулевых коэффициентов (9.9).

$$\begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix} \times \begin{pmatrix} v_{1,1} \\ v_{1,2} \\ v_{1,3} \\ v_{2,1} \\ v_{2,2} \\ v_{2,3} \\ v_{3,1} \\ v_{3,2} \\ v_{3,3} \end{pmatrix} = \begin{pmatrix} f^*_{1,1} \\ f^*_{1,2} \\ f^*_{1,3} \\ f^*_{2,1} \\ f^*_{2,2} \\ f^*_{2,3} \\ f^*_{3,1} \\ f^*_{3,2} \\ f^*_{3,3} \end{pmatrix}. \quad (9.9)$$

В монографиях Марчука и Деммеля перечислены различные прямые и итерационные методы, которые могут быть применены для решения системы (9.9), и указаны оценки их временных затрат и требования к необходимому объему оперативной памяти (табл. 9.1). Для итерационных методов используется соглашение, что временные затраты получены в предположении, что число операций итерационного процесса достаточно велико для того, чтобы снизить погрешность до некоторой фиксированной малой величины (например, до 10^{-6}). Третий и четвертый столбцы табл. 9.1 показывают оценки сложности алгоритмов (числа арифметических операций с плавающей точкой) и требуемого объема оперативной памяти (количества машинных слов), необходимого для каждого рассматриваемого метода решения СЛАУ (9.9).

Из таблицы видно, что наиболее затратным является прямой метод Холесского для плотных матриц (вариант метода исключения Гаусса для симметричных положительно определенных плотнозаполненных матриц). Наиболее предпочтительными для решения систем вида (9.9) являются итерационный многосеточный метод и прямые методы, использующие быстрое преобразование Фурье или блочную циклическую редукцию.

Таблица 9.1.

Сравнительная сложность решения уравнения Пуассона на сетке

$N_x \times N_y (n = N_x N_y)$

Метод	Прямой или итерационный	Сложность алгоритмов	Память
Холесского для плотных матриц	П	n^3	n^2
Явного обращения	П	n^2	n^2
Холесского для разреженных матриц	П	$n^{3/2}$	$n \log n$
Якоби	И	n^2	n
Зейделя	И	n^2	n
Сопряженных градиентов	И	$n^{3/2}$	n
Последовательной верхней релаксации	И	$n^{3/2}$	n
Быстрое преобразование Фурье	П	$n \log n$	n
Блочная циклическая редукция	П	$n \log n$	n
Многосеточный	И	n	n

Из рассмотренных в главе 4 итерационных методов лучшие показатели демонстрирует метод последовательной верхней релаксации. Более медленно сходящиеся методы Якоби и Зейделя требуют проведения значительного объема вычислений для достижения сходимости итерационного процесса с заданной точностью.

Методы Якоби, Зейделя и верхней релаксации

Рассмотрим применение итерационного метода Якоби для решения системы (9.8). На каждой итерации при любом порядке обхода внутренних узлов сетки при вычислении следующего приближения используется явная формула ($h_x = h_y = h$):

$$v_{i,j}^{(k+1)} = \frac{1}{4}\left(v_{i+1,j}^{(k)} + v_{i-1,j}^{(k)} + v_{i,j+1}^{(k)} + v_{i,j-1}^{(k)} - h^2 f_{i,j}\right), i, j = 1,\ldots, N; \quad k = 0,1,2,\ldots. \tag{9.10}$$

Здесь k – номер итерации, начальное приближение $\left\{v_{i,j}^{(0)}\right\}$ задано. Из литературы известно (монографии Ортеги и Деммеля), что для систем с неразложимыми матрицами, удовлетворяющих условиям диа-

гонального преобладания (что имеет место и в нашем случае), метод Якоби сходится при любом начальном приближении и любой правой части системы (9.10). Отметим перспективность применения этой формулы для параллельных вычислений, поскольку расчет каждого значения сеточной функции на новой итерации $v_{i,j}^{(k+1)}$ можно вести одновременно и независимо.

В методе Зейделя (см. главу 4) только что полученное значение $v_{i,j}^{(k+1)}$ немедленно используется в расчетах значения сеточной функции на $k+1$-й итерации в других узлах сетки. За счет такого подхода удается увеличить скорость сходимости итерационного метода, однако в этом случае существенной характеристикой полученного метода становится выбранный порядок обхода узлов сетки. Например, при обходе в сторону увеличения индексных переменных i, j получим ($h_x = h_y = h$):

$$v_{i,j}^{(k+1)} = \frac{1}{4}\left(v_{i+1,j}^{(k)} + v_{i-1,j}^{(k+1)} + v_{i,j+1}^{(k)} + v_{i,j-1}^{(k+1)} - h^2 f_{i,j}\right), i,j = 1,\ldots,N; \quad k = 0,1,2,\ldots. \tag{9.11}$$

При обходе внутренних узлов сетки в сторону уменьшения значений индексных переменных получим другой вариант метода Зейделя:

$$v_{i,j}^{(k+1)} = \frac{1}{4}\left(v_{i+1,j}^{(k+1)} + v_{i-1,j}^{(k)} + v_{i,j+1}^{(k+1)} + v_{i,j-1}^{(k)} - h^2 f_{i,j}\right), i,j = N,\ldots,1; \quad k = 0,1,2,\ldots. \tag{9.12}$$

Для рассматриваемых систем (9.9) метод Зейделя, как и метод Якоби, сходится при любом начальном приближении, поскольку матрица системы является неразложимой с диагональным преобладанием.

Заметим, что в (9.11) и (9.12) вычисления производятся по рекуррентным формулам, которые представляют определенную проблему для параллельных вычислений (см. главу 1), связанную с зависимостью вычисляемых значений сеточной функции $v_{i,j}^{(k+1)}$ не только от $\left\{v_{i,j}^{(k)}\right\}$, но и от значений сеточной функции на ($k+1$)-й итерации в других узлах сетки.

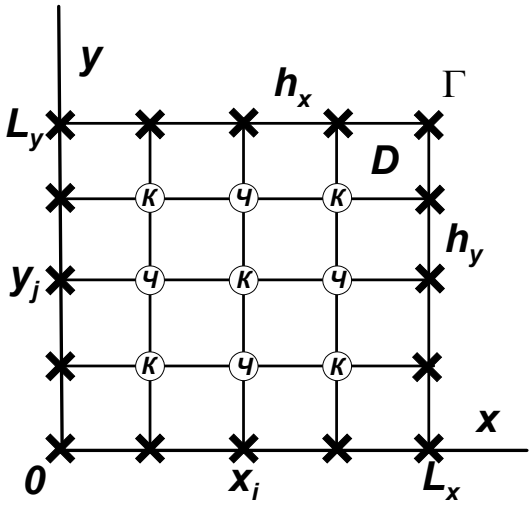


Рис. 9.3. Красно-черное упорядочивание внутренних узлов сетки

Рассмотрим перспективный для параллельной реализации метода Зейделя обход внутренних узлов сетки для итерационного решения системы (9.9), который получил название «красно-черного» или «шахматного» упорядочивания по аналогии с выбором цветов на шахматной доске. Поскольку сеточный граф внутренних вершин не содержит нечетных простых циклов, то для правильной раскраски вершин достаточно два цвета. Вариант раскраски приведен на рис. 9.3.

С учетом такого разделения множества внутренних узлов сетки ($\Omega_h = \Omega_h^B \bigcup \Omega_h^R$) алгоритм метода Зейделя преобразуется следующим образом. На каждой итерации сначала производится расчет значений сеточной функции $\left\{v_{i,j}^{(k+1)}\right\}$ в узлах одного цвета, затем – в узлах другого цвета.

Например,

- расчет значений в узлах красного цвета:

$$v_{i,j}^{(k+1)} = \frac{1}{4}\left(v_{i+1,j}^{(k)} + v_{i-1,j}^{(k)} + v_{i,j+1}^{(k)} + v_{i,j-1}^{(k)} - h^2 f_{i,j}\right), (x_i, y_j) \in \Omega_h^R; \quad (9.13)$$

- расчет значений в узлах черного цвета:

$$v_{i,j}^{(k+1)} = \frac{1}{4}\left(v_{i+1,j}^{(k+1)} + v_{i-1,j}^{(k+1)} + v_{i,j+1}^{(k+1)} + v_{i,j-1}^{(k+1)} - h^2 f_{i,j}\right), (x_i, y_j) \in \Omega_h^B. \quad (9.14)$$

Получим формулы (9.13) и (9.14) другим способом. В (9.8) или (9.9) проведем перестановку элементов в столбце неизвестных в соответствии с цветом узлов, к которым они относятся. Тогда, например, (9.9) можно переписать следующим образом ($N = 3$):

$$\begin{pmatrix}
-4 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & -4 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & -4 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & -4 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & -4 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & -4 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & -4 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & -4 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & -4
\end{pmatrix} \times \begin{pmatrix} v_{1,1} \\ v_{1,3} \\ v_{2,2} \\ v_{3,1} \\ v_{3,3} \\ v_{1,2} \\ v_{2,1} \\ v_{2,3} \\ v_{3,2} \end{pmatrix} = \begin{pmatrix} f_{1,1}^* \\ f_{1,3}^* \\ f_{2,2}^* \\ f_{3,1}^* \\ f_{3,3}^* \\ f_{1,2}^* \\ f_{2,1}^* \\ f_{2,3}^* \\ f_{3,2}^* \end{pmatrix}. \quad (9.15)$$

Из полученного представления видно, что для любого значения N в матричной форме система (9.15) будет иметь вид

$$\begin{pmatrix} D_R & C \\ C^T & D_B \end{pmatrix} \begin{pmatrix} v_R \\ v_B \end{pmatrix} = \begin{pmatrix} f_R^* \\ f_B^* \end{pmatrix}, \quad (9.16)$$

где $D_R = -4E_R, D_B = -4E_B; E_R, E_B$ – единичные матрицы порядков, равных количеству внутренних узлов красного и черного цветов. Матрица C представляет собой связи между неизвестными $\{v_{i.j}\}$ в сеточных узлах разного цвета. Применим итерации Зейделя для системы в матричной форме (9.16):

$$\begin{cases} D_R v_R^{(k+1)} + C v_B^{(k)} = f_R^* \\ C^T v_R^{(k+1)} + D_B v_B^{(k+1)} = f_B^* \end{cases}, \quad \begin{pmatrix} D_R & 0 \\ C^T & D_B \end{pmatrix} \begin{pmatrix} v_R^{(k+1)} \\ v_B^{(k+1)} \end{pmatrix} = \begin{pmatrix} f_R^* - C v_B^{(k)} \\ f_B^* \end{pmatrix} \quad (9.17)$$

или окончательно

$$\begin{cases} v_R^{(k+1)} = D_R^{-1}(f_R^* - C v_B^{(k)}) \\ v_B^{(k+1)} = D_B^{-1}(f_B^* - C^T v_R^{(k+1)}) = D_B^{-1}\left[f_B^* - C^T D_R^{-1}(f_R^* - C v_B^{(k)}) \right]. \end{cases} \quad (9.18)$$

То есть фактически при использовании красно-черного упорядочивания неизвестных в системе уравнений (9.9) итерации метода Зейделя преобразуются в последовательные итерации метода Якоби сначала для блока неизвестных значений сеточной функции в узлах красного цвета, затем для блока неизвестных в узлах черного цвета. Таким образом, применяя специальное упорядочивание неизвестных, удалось от рекуррентных вычислений (9.13) или (9.14) перейти к явным формулам (9.18), которые хорошо распараллеливаются.

Важной модификацией итерационного метода Зейделя является метод последовательной верхней релаксации, в котором при вычислении нового приближения используются результаты расчета по методу Зейделя $\left\{ \hat{v}_{i,j}^{(k+1)} \right\}$:

$$v_{i.j}^{(k+1)} = \omega \hat{v}_{i,j}^{(k+1)} + (1-\omega) v_{i,j}^{(k)}, (x_i, y_j) \in \Omega_h, k = 0,1,2,\ldots. \quad (9.19)$$

Метод (9.19) носит название метода последовательной верхней релаксации SOR, и при $0 < \omega < 2$ для систем с положительно определенной матрицей этот метод сходится. Заметим, что при $\omega = 1$ метод верхней релаксации SOR сводится к итерациям Зейделя. Из (9.19) также видно, что в зависимости от упорядочивания узлов сетки и, соответственно, уравнений можно получить различные варианты метода SOR. При красно-черном упорядочивании новые приближения к точному решению в методе последовательной верхней релаксации получаются по формулам:

- расчет значений в узлах красного цвета (см. рис. 9.3):

$$v_{i,j}^{(k+1)} = \frac{\omega}{4}\left(v_{i+1,j}^{(k)} + v_{i-1,j}^{(k)} + v_{i,j+1}^{(k)} + v_{i,j-1}^{(k)} - h^2 f_{i,j} \right) + (1-\omega) v_{i,j}^{(k)}, \quad (x_i, y_j) \in \Omega_h^R; \quad (9.20)$$

- расчет значений в узлах черного цвета:

$$v_{i,j}^{(k+1)} = \frac{\omega}{4}\left(v_{i+1,j}^{(k+1)} + v_{i-1,j}^{(k+1)} + v_{i,j+1}^{(k+1)} + v_{i,j-1}^{(k+1)} - h^2 f_{i,j}\right) + (1-\omega)v_{i,j}^{(k)}, \quad (x_i, y_j) \in \Omega_h^B. \tag{9.21}$$

Записи (9.20) и (9.21) реализуются по явным формулам, вычисления в которых производятся одновременно и независимо. Причем одна итерация метода SOR при красно-черном упорядочивании соответствует двум итерациям метода Якоби ((9.20) и (9.21)), каждая из которых применяется для обновления значений сеточной функции в узлах своего цвета.

В заключение отметим, что упорядочивание узлов сетки может быть не только бихроматическим, но и многоцветным. Это связано с выбранным шаблоном разностной схемы и хроматическим числом сеточного графа.

Параллельная реализация методов Якоби, Зейделя и верхней релаксации для решения разностных уравнений эллиптического типа

Следуя основным этапам разработки параллельных алгоритмов и программ, представленным во введении, построение параллельной версии алгоритма метода Якоби начнем с выбора фундаментальной подзадачи, затем определим коммуникационные связи между такими подзадачами, необходимые для передачи данных во время выполнения алгоритма, и найдем наиболее оптимальные способы укрупнения подзадач в соответствии с планируемым к использованию числом процессорных элементов.

В качестве мелкозернистой фундаментальной подзадачи в методе Якоби (9.10) выберем вычисление в отдельном внутреннем узле сетки значения сеточной функции приближенного решения на новой итерации $v_{i,j}^{(k+1)}$. Для расчета $v_{i,j}^{(k+1)}$ в каждой мелкозернистой подзадаче требуются значения сеточной функции на *k*-й итерации в соседних по сеточному шаблону узлах сетки (рис. 9.2). Именно это и определяет картину коммуникаций в рассматриваемой сеточной области. При объединении мелкозернистых подзадач в укрупненные блоки для рассматриваемой прямоугольной области целесообразно ввести

строки и/или столбцы фиктивных ячеек (или узлов) сетки для каждой укрупненной подзадачи. Благодаря этим фиктивным ячейкам становится возможным обновление значений сеточной функции по формуле (9.10) во внутренних узлах сетки в каждой подобласти по однородным вычислительным формулам без введения дополнительных ограничений, связанных с принадлежностью узла сеточного шаблона соседней подобласти. В то же время значения сеточной функции в фиктивных узлах должны обновляться на каждой итерации, что возможно за счет пересылки данных с соседних подобластей, где они рассчитываются.

Для рассматриваемой двумерной структурированной сетки возможны следующие способы декомпозиции (распределение узлов сетки по процессорным элементам):

- одномерная столбцовая (сеточная область разделена на подобласти вдоль оси Oy, см. рис. 9.4);
- одномерная строковая (сеточная область разделяется на подобласти вдоль оси Ox);
- двумерная (сеточная область разделена на подобласти вдоль осей Ox и Oy, см. рис. 9.4).

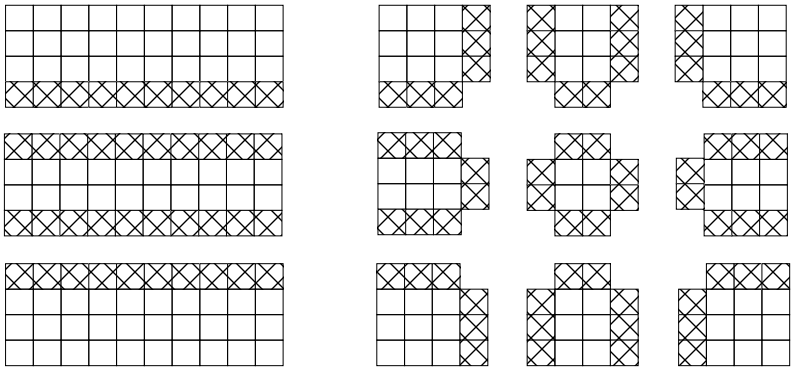


Рис. 9.4. Одномерная столбцовая (слева) и двумерная (справа) декомпозиции сеточной области ($N = 9$). Фиктивные ячейки заштрихованы

Среди одномерных схем укрупнения мелкозернистых подзадач преимуществом обладает та, которая в лучшей степени соответствует последовательной выборке данных из оперативной памяти. Сравним между собой одномерную и двумерную декомпозиции сеточной

области при использовании p процессорных элементов. Временные затраты, необходимые для проведения одной итерации без учета потерь на перемещение данных по маршруту «ОЗУ-кэш-регистры», складываются из времени вычисления значений сеточной функции во всех внутренних узлах разностной сетки и времени, затрачиваемого на пересылку данных в фиктивные узлы каждой соседней подобласти для подготовки к следующей итерации. Считая, что внутренние узлы сетки равномерно распределены по p процессорным элементам, можно оценить временные затраты при вычислениях при одномерной и двумерной декомпозиции как T_1 / p, где T_1 – время пересчета всех значений сеточной функции на одной итерации. Подсчитаем, сколько чисел будет передано/получено каждым процессорным элементом при подготовке к расчетам на следующей итерации. Для одномерной декомпозиции на каждой итерации требуется $2N$ парных пересылок для каждой подобласти, для двумерной – $4N/\sqrt{p}$ при размере сеточной области $N \times N$. Сравнивая $2N$ и $4N/\sqrt{p}$, можно отметить, что при p>4 при двумерной декомпозиции передается меньший объем информации.

Оценим временные затраты параллельного метода Якоби на одной итерации при использовании одномерной и двумерной декомпозиции:

$$T_p^{1D} \approx 5t_a \frac{N^2}{p} + 2t_{comm} \cdot 2N; \quad T_p^{2D} \approx 5t_a \frac{N^2}{p} + 2t_{comm} \cdot \frac{4N}{\sqrt{p}}; \quad T_1 \approx 5t_a \frac{N^2}{p}.$$

$$S_p^{1D} = \frac{T_1}{T_p^{1D}} \approx \frac{p}{1+\frac{4\kappa p}{5N}}; \quad S_p^{2D} = \frac{T_1}{T_p^{2D}} \approx \frac{p}{1+\frac{8\kappa\sqrt{p}}{5N}}; \quad \kappa = \frac{t_{comm}}{t_a}. \quad (9.22)$$

Здесь T_p^{1D}, T_p^{2D} – оценки временных затрат выполнения параллельного метода Якоби на каждой итерации при одномерной и двумерной декомпозиции сеточной области.

Построенные по формулам (9.22) графики оценки ускорения показывают (рис. 9.5), что при фиксированном размере задачи двумерная декомпозиция имеет значительное преимущество при использовании большего числа процессорных элементов.

Поскольку итерации метода Зейделя и последовательной верхней релаксации при красно-черном упорядочивании сводятся к итерациям метода Якоби, то и распараллеливание этих методов осуществляется аналогично распараллеливанию метода Якоби.

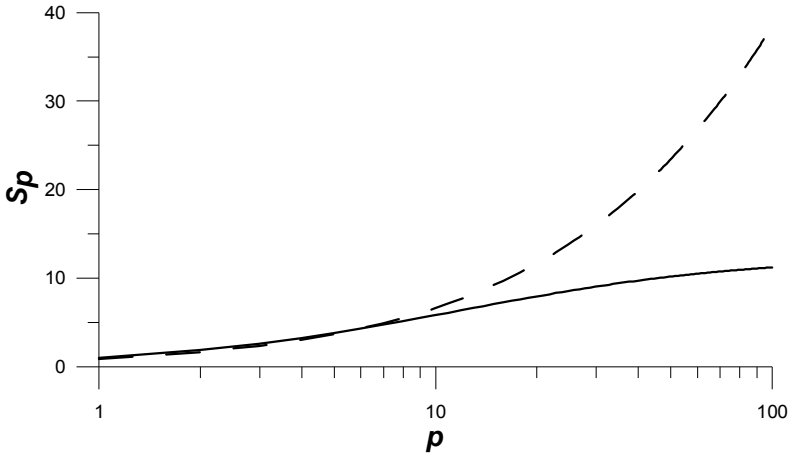


Рис. 9.5. Графики оценки ускорения параллельного метода Якоби при одномерной и двумерной декомпозиции при фиксированном размере задачи (n = 100, к = 10). Сплошная линия – одномерная декомпозиция, штриховая – двумерная

9.2. Параллельные алгоритмы решения задачи нестационарной теплопроводности с помощью явных и неявных разностных схем

Рассмотрим задачу о распространении тепла в бесконечном твердом бруске квадратного поперечного сечения, на боковых стенках которого заданы значения температуры, не меняющиеся по длине бруска. В начальный момент времени температура бруска является постоянной. В случае, если коэффициент температуропроводности a^2 постоянен, рассматриваемый физический процесс имеет двумерную картину и в каждом поперечном сечении бруска $\bar{D} = [0, L] \times [0, L]$ описывается уравнением теплопроводности с краевыми условиями вида:

$$\begin{cases} \dfrac{\partial u}{\partial t} = a^2\left(\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2}\right), (x,y) \in D, t > 0; \\ u(0,x,y) = u_0, (x,y) \in \bar{D}; \\ u(t,x,y) = u_B(t,x,y), (x,y) \in \Gamma = \bar{D} \setminus D, t > 0. \end{cases} \tag{9.24}$$

Приближенное решение задачи (9.24) будем искать с использованием метода конечных разностей. Для области $\bar{D}$ построим равномерную прямоугольную сетку (см. рис. 9.1):

$$\bar{\omega}_h = \left\{ \begin{array}{l} x_i = h_x \cdot i; \;\; i = 0,...,N_x + 1; h_x = L/\left(N_x + 1\right) \\ y_j = h_y \cdot j; j = 0,...,N_y + 1; h_y = L/\left(N_y + 1\right) \end{array} \right\}.$$

Для перехода от дифференциальной постановки (9.24) к конечно-разностной введем сеточную функцию $\left\{v_{i,j}^m\right\}$: $v_{i,j}^m \approx u(t^m, x_i, y_j), (x_i, y_j) \in \bar{\omega}_h, t^m = m \cdot \tau, \; m = 0,1,2,...$. Сеточная функция $v_{i,j}^n$ зависит от индексов узлов сетки i, j и номеров временных слоев m. Найденные значения сеточной функции будут представлять приближенное решение задачи (9.24) в выбранных заранее узлах сетки в определенные моменты времени.

Перейдем к построению разностной схемы для уравнения (9.24). Для этого рассмотрим это уравнение во внутреннем узле сетки $(x_i, y_j) \in \omega_h$ в момент времени t^m. В качестве разностного шаблона выберем шаблон, представленный на рис. 9.6.

Приведем конечно-разностные формулы для аппроксимации дифференциальных операторов уравнения теплопроводности:

$$\left(\frac{\partial u}{\partial t}\right)_{i,j}^m \approx \frac{v_{i,j}^{m+1} - v_{i,j}^m}{\tau};$$

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j}^m \approx \sigma\left(\frac{v_{i+1,j}^{m+1} - 2v_{i,j}^{m+1} + v_{i-1,j}^{m+1}}{h_x^2}\right) + (1-\sigma)\left(\frac{v_{i+1,j}^m - 2v_{i,j}^m + v_{i-1,j}^m}{h_x^2}\right);$$

$$\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j}^{m} \approx \sigma\left(\frac{v_{i,j+1}^{m+1} - 2v_{i,j}^{m+1} + v_{i,j-1}^{m+1}}{h_y^2}\right) + (1-\sigma)\left(\frac{v_{i,j+1}^{m} - 2v_{i,j}^{m} + v_{i,j-1}^{m}}{h_y^2}\right).$$

Здесь σ – весовой коэффициент ($0 \le \sigma \le 1$).

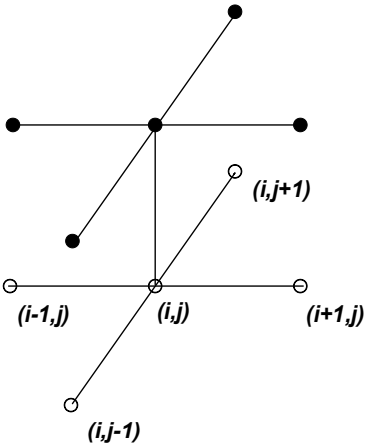


Рис. 9.6. Разностный шаблон для уравнения (9.24).
Светлые значки относятся к *m*-му слою по времени,
темные – к *(m+1)*-му

Применив рассмотренные выше разностные аппроксимации дифференциальных операторов из (9.24) во внутренних узлах сетки и добавив к полученным разностным уравнениям начальные и граничные условия в соответствующих узлах сетки, запишем систему линейных алгебраических уравнений (9.25).

Весовой коэффициент σ определяет вид разностной схемы: если $\sigma = 0$, то схема явная, т.е. представляет собой разрешенные относительно неизвестных $\left\{v_{i,j}^{m+1}\right\}$ уравнения; если $0 < \sigma \le 1$, то схема неявная, в этом случае требуется дополнительно решать уравнения разностной схемы, чтобы найти значения неизвестных $\left\{v_{i,j}^{m+1}\right\}$. Среди неявных схем особого внимания заслуживает схема Кранка – Николсон ($\sigma = 0{,}5$). Эта разностная схема имеет второй порядок аппроксимации по времени и координатным направлениям.

$$
\begin{cases}
\dfrac{v_{i,j}^{m+1}-v_{i,j}^{m}}{\tau}= \\
=a^2\left[\sigma\left(\dfrac{v_{i+1,j}^{m+1}-2v_{i,j}^{m+1}+v_{i-1,j}^{m+1}}{h_x^2}\right)+(1-\sigma)\left(\dfrac{v_{i+1,j}^{m}-2v_{i,j}^{m}+v_{i-1,j}^{m}}{h_x^2}\right)\right]+ \\
+a^2\left[\sigma\left(\dfrac{v_{i,j+1}^{m+1}-2v_{i,j}^{m+1}+v_{i,j-1}^{m+1}}{h_y^2}\right)+(1-\sigma)\left(\dfrac{v_{i,j+1}^{m}-2v_{i,j}^{m}+v_{i,j-1}^{m}}{h_y^2}\right)\right]; \\
\quad i=\overline{1,N_x};\ j=\overline{1,N_y};\ m=0,1,2,... \\
\\
v_{0,j}^{m+1}=u_B(0,y_j);\quad v_{N_x+1,j}^{m+1}=u_B(L,y_j);\quad j=\overline{0,N_y+1};\ m=0,1,2,... \\
v_{i,0}^{m+1}=u_B(x_i,0);\quad v_{i,N_y+1}^{m+1}=u_B(x_i,L);\quad i=\overline{0,N_x+1};\ m=0,1,2,... \\
v_{i,j}^{0}=u_0(x_i,y_j);\quad i=\overline{0,N_x+1};\ j=\overline{0,N_y+1}.
\end{cases}
\tag{9.25}
$$

При $\sigma = 0$ легко получить явную формулу для непосредственного вычисления значения $\left\{v_{i,j}^{m+1}\right\}$. При $\sigma > 0$ (9.25) представляет собой систему линейных уравнений с пятидиагональной матрицей для вектора неизвестных $\left\{v_{i,j}^{m+1}\right\}$, решив которую можно найти неизвестные, поэтому при $\sigma > 0$ разностная схема (9.25) называется неявной.

Отметим некоторые свойства полученной разностной схемы при $\sigma = 0$ и $\sigma = 1$. Явная разностная схема ($\sigma = 0$) является условно устойчивой по начальным данным[‡], что накладывает ограничения на величину шага интегрирования уравнения по времени:

$$\tau < \left(2a^2(h_x^{-2}+h_y^{-2})\right)^{-1}. \tag{9.26}$$

Из (9.26) видно, что шаг по времени тем меньше, чем больше значение коэффициента температуропроводности и меньше шаг разностной сетки $\overline{\omega}_h$.

[‡] Разностная схема называется устойчивой по начальным данным, если малые вариации начальных данных ведут к малым изменениям численного решения.

Неявная разностная схема (σ=1) абсолютно устойчива, что теоретически не требует учета какого-либо ограничения на шаг τ.

Эти схемы аппроксимируют дифференциальное уравнение с первым порядком по времени и вторым по пространству.

Явная схема

Из (9.25) при $\sigma = 0$ получим

$$v_{i,j}^{m+1} = v_{i,j}^{m} + \tau \cdot a^2 \left(\frac{v_{i+1,j}^{m} - 2v_{i,j}^{m} + v_{i-1,j}^{m}}{h_x^2} \right) + \\ + \tau \cdot a^2 \left(\frac{v_{i,j+1}^{m} - 2v_{i,j}^{m} + v_{i,j-1}^{m}}{h_y^2} \right); i = \overline{1, N_x}; \quad j = \overline{1, N_y}. \tag{9.27}$$

Перепишем полученную формулу в следующем виде:

$$v_{i,j}^{m+1} = (1 - ap) v_{i,j}^{m} + ae \cdot v_{i+1,j}^{m} + aw \cdot v_{i-1,j}^{m} + an \cdot v_{i,j+1}^{m} + as \cdot v_{i,j-1}^{m}; \\ i = \overline{1, N_x}; \quad j = \overline{1, N_y}. \tag{9.28}$$

Коэффициенты *ap*, *ae*, *aw*, *an*, *as* определяются как:

$$ae = aw = \frac{a^2 \cdot \tau}{h_x^2}; \; an = as = \frac{a^2 \cdot \tau}{h_y^2}; \quad ap = ae + aw + an + as.$$

Заметим, что вычисления по явным формулам (9.27) или (9.28) можно проводить одновременно и независимо, что открывает большие возможности при построении параллельных алгоритмов для решения эволюционных уравнений теплопроводности.

Следуя основным этапам разработки параллельных алгоритмов, подробно рассмотренным во введении, произведем декомпозицию задачи расчета температуры $\left\{ v_{i,j}^{m+1} \right\}$ на (*m*+*1*)-м слое во внутренних узлах. В качестве фундаментальной мелкозернистой подзадачи выберем задачу расчета одного значения $v_{i,j}^{m+1}$ для фиксированных значений индексов *i* и *j* по формуле (9.28). Общее количество таких фундаментальных подзадач будет $N_x \times N_y$, что совпадает с количеством внутренних узлов разностной сетки и позволяет декомпозиции

вычислительной задачи поставить в соответствие декомпозицию сеточной области.

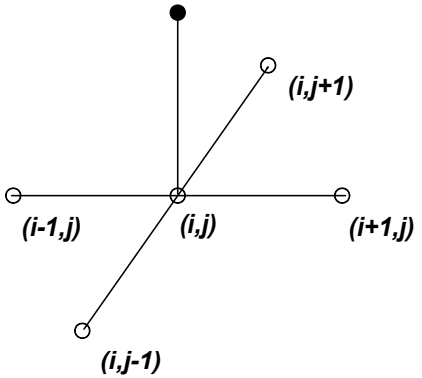


Рис. 9.7 Шаблон явной разностной схемы. Светлые значки относятся к *m*-му слою по времени, темные – к *(m+1)*-му

Взаимосвязь фундаментальных подзадач на этапе проектирования коммуникаций определяется выбранным сеточным шаблоном (рис. 9.7), в соответствии с которым для вычисления $v_{i,j}^{m+1}$ требуются значения $v_{i-1,j}^{m}, v_{i+1,j}^{m}, v_{i,j-1}^{m}, v_{i,j+1}^{m}, v_{i,j}^{m}$. На этапе укрупнения производится объединение фундаментальных подзадач или внутренних узлов сетки, для которых выполняются эти подзадачи, в подобласти. Поскольку массив фундаментальных подзадач представляет собой матрицу размером $N_x \times N_y$, возможно как одномерное деление сеточной области на подобласти, так и двумерное – главное, чтобы входящие в подобласть узлы сетки представляли собой упорядоченную структуру. Для обеспечения однородности вычислений целесообразно ввести дополнительные фиктивные узлы разностной сетки на границах декомпозиции сеточной области, значения температуры в которых будут рассчитываться другим процессорным элементом в соседней подобласти (см. рис. 9.4, 9.8).

Таким образом, на каждом временном слое рассматриваемого параллельного алгоритма решения уравнения теплопроводности с по-

мощью явной разностной схемы выполняются две следующие операции:

- процессорные элементы одновременно ведут расчеты значений $\left\{v_{i,j}^{m+1}\right\}$, каждый внутри своей сеточной подобласти;

- для подготовки данных для расчета температуры на следующем временном слое в каждой подобласти сетки производится передача в фиктивные узлы значений температуры, рассчитанных в соседней сеточной подобласти.

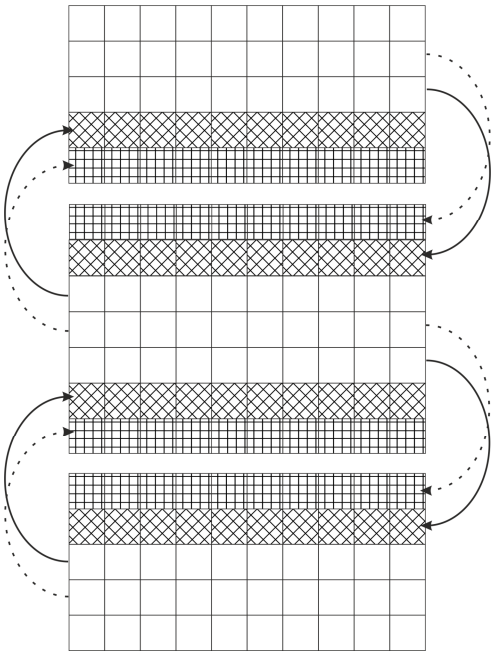


Рис. 9.8. Декомпозиция сеточной области для трех процессорных элементов. Заштрихованы фиктивные ячейки. Расчет $v_{i,j}^{m+1}$ проводится в незаштрихованных и диагонально заштрихованных ячейках. Расчет $v_{i,j}^{m+2}$ – только в незаштрихованных ячейках. Стрелками показана межпроцессорная передача данных

В целом, можно сказать, что стратегия распараллеливания явных разностных схем во многом совпадает с построением параллельного

алгоритма итерационного метода Якоби для решения разностной задачи Дирихле для уравнения Пуассона, подробно рассмотренного в п. 9.1.

В заключение остановимся на описании одного способа сокращения объема временных затрат на межпроцессорную передачу данных во время проведения расчетов по явной разностной схеме. Это достигается за счет увеличения количества сеточных линий, содержащих фиктивные узлы вблизи границ декомпозиции сеточной области, что позволяет пересылать данные не на каждом временном шаге параллельного алгоритма, а через несколько шагов. Правда, и объем формируемых для межпроцессорной передачи данных возрастает в такое же число раз.

На рис. 9.8 представлена одномерная декомпозиция сеточной области для трех процессорных элементов. Вместо одной сеточной линии с фиктивными узлами на границе каждой подобласти используется две. В фиктивные узлы с соседнего процессорного элемента копируются значения сеточной функции $\left\{v_{i,j}^{m}\right\}$. Наличие дополнительных фиктивных узлов позволяет без межпроцессорной передачи данных рассчитать во внутренних узлах каждой подобласти значения $\left\{v_{i,j}^{m+1}\right\}$ и $\left\{v_{i,j}^{m+2}\right\}$. Сокращение временных затрат на передачу данных от одного процессорного элемента другому обеспечивается балансом между объемом передаваемых данных, частотой их передачи, разрешенной длиной передаваемого сообщения в стандарте Message Passing Interface.

Неявная схема

При $\sigma = 1$ из (9.25) получаем:

$$v_{i,j}^{m+1} = v_{i,j}^{m} + \tau \cdot a^2 \left(\frac{v_{i+1,j}^{m+1} - 2v_{i,j}^{m+1} + v_{i-1,j}^{m+1}}{h_x^2} \right) + \tau \cdot a^2 \left(\frac{v_{i,j+1}^{m+1} - 2v_{i,j}^{m+1} + v_{i,j-1}^{m+1}}{h_y^2} \right); i = \overline{1, N_x}; \quad j = \overline{1, N_y}; \quad m = 0,1,2,...; \tag{9.29}$$

$$v_{0,j}^{m+1} = u_B\left(0, y_j\right); v_{N_x+1,j}^{m+1} = u_B\left(L, y_j\right); m = 0,1,2,...; j = 0,...,N_y + 1;$$
$$v_{i,0}^{m+1} = u_B\left(x_i, 0\right); v_{i,N_y+1}^{m+1} = u_B\left(x_i, L\right); m = 0,1,2,...; i = 0,...,N_x + 1;$$
$$v_{i,j}^{0} = u_0(x_i, y_j);\ i = 0,...,N_x + 1;\ \ j = 0,...,N_y + 1.$$

Перепишем полученные формулы в следующем виде:

$$(1 + ap)v_{i,j}^{m+1} = ae \cdot v_{i+1,j}^{m+1} + aw \cdot v_{i-1,j}^{m+1} + an \cdot v_{i,j+1}^{m+1} + as \cdot v_{i,j-1}^{m+1} + b_{i,j};$$
$$i = 0,..., N_x + 1; j = 0,..., N_y + 1. \qquad (9.30)$$

Коэффициенты $ap, ae, aw, an, as, b_{i,j}$ определяются из (9.30):

$$ae = aw = \frac{a^2 \cdot \tau}{h_x^2};\ an = as = \frac{a^2 \cdot \tau}{h_y^2}; ap = ae + aw + an + as;$$
$$b_{i,j} = v_{i,j}^m; i = \overline{1, N_x};\ \ j = \overline{1, N_y};$$
$$i = 0; ap = 1;\ b_{0,j} = u_B(0, y_j); ae = aw = as = an = 0;$$
$$i = N_x + 1;\ \ ap = 1;\ \ b_{N_x+1,j} = u_B(L, y_j);\ \ ae = aw = as = an = 0;$$
$$j = 0; ap = 1;\ b_{i,0} = u_B(x_i, 0); ae = aw = as = an = 0;$$
$$j = N_y + 1;\ \ ap = 1;\ b_{i,N_y+1} = u_B(x_i, L);\ ae = aw = as = an = 0.$$

Разностная схема (9.30) представляет собой систему линейных уравнений вида $A\vec{x} = \vec{b}$, в которой $\vec{x}$ есть вектор-столбец неизвестных, компоненты которого – значения сеточной функции $\left\{v_{i,j}^{m+1}\right\}$. На рис. 9.9 приведен вид вектора $\vec{x}$, длина которого составляет $(N_x + 2)(N_y + 2)$.

Матрица A системы (9.30) есть пятидиагональная симметричная, положительно определенная матрица, состоящая из блоков четырех видов (рис. 9.10): блоков *E*, соответствующих единичной матрице, блоков *T* с трехдиагональной матрицей, блоков *K*, с диагональной матрицей, в которой первая и последние строки состоят только из нулевых элементов, и блоков *0*, объединяющих нулевые значения.

$$\vec{x} = \begin{pmatrix} v_{0,0}^{m+1} \\ v_{1,0}^{m+1} \\ v_{2,0}^{m+1} \\ \vdots \\ v_{N_x+1,0}^{m+1} \\ v_{0,1}^{m+1} \\ v_{1,1}^{m+1} \\ v_{2,1}^{m+1} \\ v_{3,1}^{m+1} \\ \vdots \\ v_{N_x+1,N_y+1}^{m+1} \end{pmatrix}; \quad \vec{b} = \begin{pmatrix} u_B(x_0, y_0) \\ u_B(x_1, y_0) \\ u_B(x_2, y_0) \\ \vdots \\ u_B(x_{N_x+1}, y_0) \\ u_B(x_0, y_1) \\ v_{1,1}^{m} \\ v_{2,1}^{m} \\ v_{3,1}^{m} \\ \vdots \\ u_B(x_{N_x+1}, y_{N_y+1}) \end{pmatrix}$$

Рис. 9.9. Представление вектора-столбца неизвестных и вектора-столбца правой части системы (9.30)

$$A = \begin{bmatrix} E & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ K & T & K & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & K & T & K & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & K & T & K & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & K & T & K & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & K & T & K & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & K & T & K & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & K & T & K & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & K & T & K & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & K & T & K \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & E \end{bmatrix}$$

Рис. 9.10. Вид матрицы A системы (9.30)

Для численного решения на МВС с распределенной памятью систем линейных алгебраических уравнений вида (9.30), полученных

после применения неявных аппроксимационных формул для дифференциальной постановки задачи, как правило, не используются прямые методы в силу невысокой эффективности их параллельных версий (метода прогонки и метода циклической редукции, см. п.3.3), чувствительности к влиянию погрешности округления, что особенно проявляется при решении плохо обусловленных систем большой размерности, к которым относится и задача (9.30). Поэтому имеет смысл на каждом шаге по времени для решения системы (9.30) применять итерационные методы. Приведенные в табл. 9.1 результаты сравнительного анализа сложности различных методов решения разностной задачи Дирихле для уравнения Пуассона позволяют сделать вывод в пользу итерационных методов последовательной верхней релаксации, сопряженных градиентов и многосеточного метода.

Метод сопряженных градиентов

Одними из перспективных с точки зрения обеспечения высокой скорости сходимости итерационных методов решения плохо обусловленных систем вида $A\vec{x} = \vec{b}$ с разреженной матрицей A являются методы вариационного типа (например, стабилизированный метод бисопряженных градиентов BiCGStab, обобщенный метод минимальных невязок GMRES и др.), которые строятся на основе выбора итерационных параметров метода из требования минимизации функционалов, определяющих степень близости текущего приближения к точному решению системы уравнений (см. п.4.3).

В данном разделе рассмотрим более простой в реализации метод сопряженных градиентов, для применимости которого требуется, чтобы матрица A была симметричной и положительно определенной, что и имеет место для случая неявной разностной схемы (9.30).

Для метода сопряженных градиентов, как и для других градиентных методов (см. п. 4.3), во время итерационного процесса производится минимизация функционала

$$\varphi(\vec{x}) = \frac{1}{2}\left(\vec{x}, A\vec{x}\right) - \left(\vec{x}, \vec{b}\right)$$

за счет получения приближенного решения рассматриваемой задачи по итерационной формуле

$$\vec{x}_{k+1} = \vec{x}_k - \alpha_k \vec{p}_k, \quad k = 0,1,2,\ldots, \ \vec{x}_0 \ - \text{задано};$$

в которой параметры α_k выбираются из условия минимума $\varphi(\vec{x}_k - \alpha_k \vec{p}_k)$:

$$\alpha_k = -\frac{(\vec{p}_k, \vec{r}_k)}{(\vec{p}_k, A\vec{p}_k)},$$

а векторы сопряженных направлений генерируются как

$$\vec{p}_{k+1} = \vec{r}_{k+1} + \beta_k \vec{p}_k, \quad k = 0,1,2,...; \vec{p}_0 = \vec{r}_0 = \vec{b} - A\vec{x}_0.$$

Здесь коэффициенты $\{\beta_k\}$ рассчитываются по формуле, полученной с использованием свойств векторов $\{\vec{r}_k\}$ и $\{\vec{p}_k\}$:

$$\beta_k = -\frac{(\vec{p}_k, A\vec{r}_{k+1})}{(\vec{p}_k, A\vec{p}_k)} = \frac{(\vec{r}_{k+1}, \vec{p}_{k+1})}{(\vec{r}_k, \vec{p}_k)} = \frac{(\vec{r}_{k+1}, \vec{r}_{k+1})}{(\vec{r}_k, \vec{r}_k)}.$$

В окончательном виде метод сопряженных градиентов представляется следующим образом:

- ***выбрать*** $\vec{x}_0$***, вычислить*** $\vec{r}_0 = \vec{b} - A\vec{x}_0$***, положить*** $\vec{p}_0 = \vec{r}_0$***,***
- ***рассчитать*** $(\vec{r}_0, \vec{r}_0)$***,***
- ***повторять для*** $k = 0,1,2,...$

  $\alpha_k = -(\vec{r}_k, \vec{r}_k) / (\vec{p}_k, A\vec{p}_k)$,

  $\vec{x}_{k+1} = \vec{x}_k - \alpha_k \vec{p}_k$,

  $\vec{r}_{k+1} = \vec{r}_k + \alpha_k A\vec{p}_k$,

- ***рассчитать*** $(\vec{r}_{k+1}, \vec{r}_{k+1})$ ***и если*** $\sqrt{(\vec{r}_{k+1}, \vec{r}_{k+1})} \geq \varepsilon$***, то продолжать***,

  $\beta_k = (\vec{r}_{k+1}, \vec{r}_{k+1}) / (\vec{r}_k, \vec{r}_k)$,

  $\vec{p}_{k+1} = \vec{r}_{k+1} + \beta_k \vec{p}_k$.

- ***конец цикла***

Скорость сходимости метода сопряженных градиентов на каждой итерации можно оценить по следующей формуле:

$$\left\| \vec{x}_k - \vec{x}^* \right\|_2 \leq 2\sqrt{\nu}\delta^k \left\| \vec{x}_k - \vec{x}^* \right\|_2,$$

где $v = cond(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \lambda_{\max} / \lambda_{\min}$ – число обусловленности матрицы A; $\lambda_{\max}, \lambda_{\min}$ – наибольшее и наименьшее собственные значения матрицы; $\|\vec{x}\|_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$; $\delta = \dfrac{\sqrt{v}-1}{\sqrt{v}+1}$; k – номер итерации; $\vec{x}^*$ – точное решение;

Если $v \sim 1, \delta \ll 1$, то сходимость метода сопряженных градиентов быстрая. Если $v \gg 1, \delta \sim 1$ (что свойственно для плохо обусловленных систем), то сходимость метода сопряженных градиентов медленная. Это наблюдение приводит к общему понятию предобусловливания матрицы A посредством преобразования конгруэнтности $\tilde{A} = SAS'$, где S – невырожденная матрица, выбранная так, чтобы

$$cond(\tilde{A}) < cond(A).$$

Тогда, используя матрицу S, можно осуществить переход от решения системы $A\vec{x} = \vec{b}$ к решению системы $\tilde{A}\tilde{\vec{x}} = \tilde{\vec{b}}$, где $\tilde{A} = SAS'$, $\tilde{\vec{x}} = (S')^{-1}\vec{x}$, $\tilde{\vec{b}} = S\vec{b}$. Матрица $\tilde{A}$ также остается положительно определенной матрицей, как и A, однако ее число обусловленности меньше, что позволяет надеяться на сокращение числа итераций при использовании предобусловленного метода сопряженных градиентов.

Для предобусловливания систем линейных алгебраических уравнений часто используют:

- диагональные или блочно-диагональные матрицы,
- метод последовательной релаксации SSOR или метод Якоби с ограниченным числом итераций,
- методы неполной факторизации,
- полиномиальное предобусловливание.

Поскольку при построении параллельных версий предобусловленного метода сопряженных градиентов не все предобусловливатели эффективно распараллеливаются, необходимо специальное внимание уделять их выбору.

Рассмотрим подход предобусловливания, основанный на использовании фиксированной симметричной положительно определенной матрицы $M = (S'S)^{-1}$. Основными требованиями выбора матрицы M

являются простота решения системы $M\tilde{\vec{r}}_k = \vec{r}_k$ и чтобы матрица M хорошо аппроксимировала матрицу A.

Окончательно предобусловленный метод сопряженных градиентов имеет вид:

- выбрать $\vec{x}_0$***, вычислить*** $\vec{r}_0 = \vec{b} - A\vec{x}_0$***, решить систему*** $M\tilde{\vec{r}}_0 = \vec{r}_0$***,***

- положить $\vec{p}_0 = \tilde{\vec{r}}_0$***,***

- рассчитать $\left(\tilde{\vec{r}}_0, \vec{r}_0\right)$***,***

- повторять для $k = 0,1,2,...$

$$\alpha_k = -\left(\tilde{\vec{r}}_k, \vec{r}_k\right) / \left(\vec{p}_k, A\vec{p}_k\right),$$

$$\vec{x}_{k+1} = \vec{x}_k - \alpha_k \vec{p}_k,$$

$$\vec{r}_{k+1} = \vec{r}_k + \alpha_k A\vec{p}_k,$$

- рассчитать $\left(\vec{r}_{k+1}, \vec{r}_{k+1}\right)$ ***и если*** $\sqrt{\left(\vec{r}_{k+1}, \vec{r}_{k+1}\right)} \geq \varepsilon$***,***

то продолжать,

- решить систему $M\tilde{\vec{r}}_{k+1} = \vec{r}_{k+1}$***,***

$$\beta_k = \left(\tilde{\vec{r}}_{k+1}, \vec{r}_{k+1}\right) / \left(\tilde{\vec{r}}_k, \vec{r}_k\right),$$

$$\vec{p}_{k+1} = \tilde{\vec{r}}_{k+1} + \beta_k \vec{p}_k.$$

- конец цикла

Метод сопряженных градиентов имеет то особенно привлекательное свойство, что в его реализации предусмотрено одновременное хранение в памяти лишь четырех векторов: $\vec{x}_k, \vec{r}_k, \vec{p}_k, \vec{v}_k = A\vec{p}_k$ (пяти – для предобусловленного метода сопряженных градиентов).

Кроме того, в его внутреннем цикле, помимо матрично-векторного произведения (и решения системы уравнений $M\tilde{\vec{r}}_{k+1} = \vec{r}_{k+1}$ для предобусловленного метода сопряженных градиентов), вычисляются только два скалярных произведения, три операции типа «**saxpy**» (сложение вектора, умноженного на число, с другим вектором) и небольшое количество скалярных операций. Таким образом, необходимые ресурсы оперативной памяти и объем вычислительной работы в методе не очень велики.

Чтобы яснее представить себе, как следует провести построение параллельного алгоритма метода сопряженных градиентов, разберем отдельно каждую операцию между матрицей A и векторами $\vec{x}_k, \vec{r}_k, \vec{p}_k$, которые используются в представленном выше алгоритме.

Первая операция – это матрично-векторное произведение $\vec{v}_k = A\vec{p}_k$. Известно (см. также главу 2), что в общем случае при использовании параллельного алгоритма умножения матрицы на вектор возможны два способа распределения данных между p процессорными элементами ($p \ll N_x, p \ll N_y$).

В первом случае каждому процессорному элементу назначается определенное количество строк матрицы A и целиком вектор $\vec{x}$ (рис. 9.11). Каждый процессорный элемент выполняет умножение распределенных ему строк матрицы A на вектор $\vec{p}_k$. Получаем абсолютно не связанные между собой подзадачи, однако предложенный параллельный алгоритм не лишен и недостатков. Дело в том, что компоненты результирующего вектора $\vec{v}_k$ оказываются распределенными по всем процессорным элементам и для выполнения следующей итерации метода сопряженных градиентов требуется собрать вектор $\vec{v}_k$ на каждом процессорном элементе.

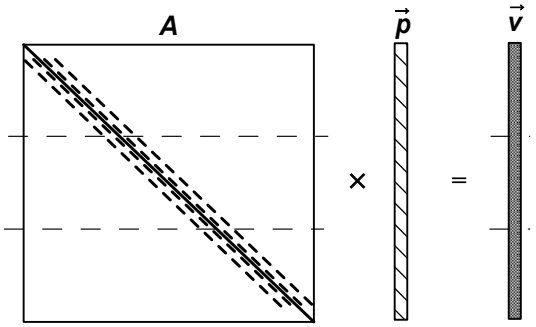


Рис. 9.11. Строчное распределение данных

Другой вариант – каждому процессорному элементу распределяется определенное количество столбцов матрицы A, поэтому нет необходимости хранить на каждом процессоре вектор $\vec{p}_k$ целиком, и он распределяется по всем процессорным элементам (схема распре-

деления данных представлена на рис. 9.12). Тогда в процессе вычислений результирующий вектор $\vec{v}_k$ оказывается разделенным на векторные слагаемые, где каждое векторное слагаемое находится на отдельном процессорном элементе. Поэтому вновь потребуются пересылки данных для того, чтобы собрать результирующий вектор $\vec{v}_k$ и разослать каждому процессорному элементу часть вектора $\vec{v}_k$, необходимую для начала следующей итерации.

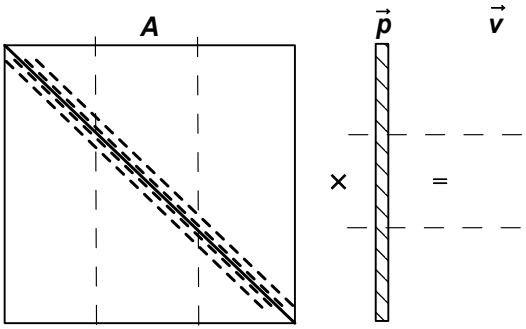


Рис. 9.12. Столбцовое распределение данных

Для случая системы с ленточной матрицей более эффективным будет способ столбцового распределения данных по процессорным элементам, так как слагаемые, из которых складываются компоненты результирующего вектора $\vec{v}_k$, часто будут отличны от нуля лишь на одном процессорном элементе, т.е. сокращается количество межпроцессорных обменов.

Следующие две операции – это линейная комбинация векторов **saxpy** и вычисление скалярного произведения (рис. 9.13). При параллельном сложении векторов каждый процессорный элемент выполняет действия над распределенными ему компонентами векторов, эта операция не требует обменов данными, если нет необходимости в последующем использовании вектора целиком (см. Введение).

Параллельное вычисление скалярного произведения – несколько другая операция в том смысле, что ее результатом является не вектор, а одно число, которое впоследствии должно быть отправлено на все вычислительные узлы для вычисления параметров метода или проверки условия завершения итераций. При вычислении скалярного

произведения первый этап – покомпонентное умножение векторов, в результате которого получается вектор, – происходит независимо на каждом процессорном элементе подобно сложению векторов. Затем необходимо просуммировать все компоненты полученного вектора. Сначала на каждом процессорном элементе выполняется суммирование принадлежащих ему компонент вектора, затем результаты сложения собираются и суммируются на одном из процессорных элементов, полученный результат рассылается остальным вычислительным узлам.

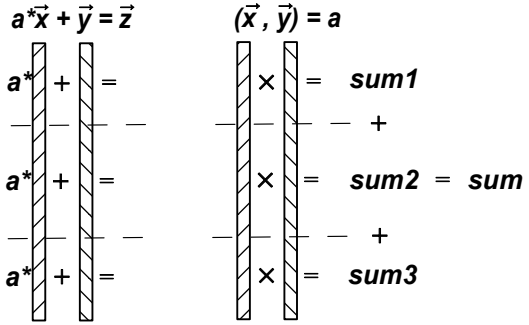


Рис. 9.13. Схема операций saxpy и скалярного произведения

Тестовые расчеты, проведенные по неявной разностной схеме методом сопряженных градиентов с фиксированным шагом по времени для различных значений коэффициента температуропроводности, показали незначительное увеличение времени счета с ростом a^2. Это объясняется небольшим возрастанием числа итераций, требуемых для сходимости итерационного метода сопряженных градиентов на каждом временном шаге с заданной точностью ε. Рис. 9.14 показывает, сколько итераций метода необходимо для сходимости вычислительного процесса на каждом временном шаге. Кроме того, применение неявной схемы при больших значениях параметра a^2 имеет преимущество по сравнению с явными разностными схемами, поскольку не требует существенного ограничения величины шага по времени.

В табл. 9.2 представлено время выполнения параллельных программ, использующих одномерную декомпозицию, при решении нестационарного уравнения теплопроводности с помощью явной и неявной разностных схем.

Рис. 9.14 Количество локальных итераций метода сопряженных градиентов на каждом шаге по времени, $\tau = 0{,}6$ с; $h = 0{,}00347$ м, $a^2 = 10^{-4}$ м^2/с, $\varepsilon = 10^{-5}$

Таблица 9.2

Время работы параллельной программы решения уравнения теплопроводности для $0 < t \le 100$ с помощью неявной и явной разностных схем, с

Число процессорных элементов	1	4	9
Неявная схема	112	56	36
Явная схема	318	89	48

Для принятого значения коэффициента температуропроводности $a^2 = 10^{-4}$ м^2/с и выбранной сетки 288x288 шаг интегрирования по времени для явной разностной схемы составил величину 0,012 с. Неявная разностная схема применялась с шагом 0,6 с. Из таблицы видно, что в этом случае применение неявной схемы и метода сопряженных градиентов имеет преимущество, которое, однако, уменьшается с ростом числа используемых процессорных элементов, что связано с увеличением затрат на межпроцессорную передачу данных.

Литература

1. *Хокни Р., Джесссхоуп К.* Параллельные ЭВМ. Архитектура, программирование и алгоритмы. М.: Радио и связь, 1986.
2. *Ортега Дж.* Введение в параллельные и векторные методы решения линейных систем. М.: Мир, 1991.
3. *Немнюгин С.А., Стесик О.Л.* Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.
4. *Гергель В.П., Стронгин Р.Г.* Основы параллельных вычислений для многопроцессорных вычислительных машин. Нижний Новгород: Изд-во ННГУ им. Н.И.Лобачевского, 2000.
5. *Старченко А.В., Есаулов А.О.* Параллельные вычисления на многопроцессорных вычислительных системах. Томск: Изд-во Том. ун-та, 2002.
6. *Голуб Дж., Ван Лоун Дж.* Матричные вычисления. М.: Мир, 1999.
7. *Деммель Дж.* Вычислительная линейная алгебра. М.: Мир, 2001.
8. *Самарский А.А.* Введение в численные методы. СПб.: Лань, 2005.
9. *Самарский А.А., Николаев Е.С.* Методы решения сеточных уравнений. М.: Наука, 1978. 591 с.
10. *Яненко Н.Н., Коновалов А.Н., Бугров А.Н., Шустов Г.В.* Об организации параллельных вычислений и распараллеливании прогонки // Численные методы механики сплошных сред. 1978. №7. С.136–139.
11. *Вшивков В.А.* О распараллеливании вычислительных алгоритмов // Сибирская школа-семинар по параллельным вычислениям. Томск: Изд-во Том. ун-та, 2002. С.46–59.
12. *Ильин В.П.* Методы конечных разностей и конечных объемов для эллиптических уравнений. Новосибирск: Изд-во Института математики, 2000.
13. *Duff L.S., VanderVorst H.A.* Development and trends in the parallel solution of linear systems // Parallel Computing, 1999, Vol.25. P.1931–1970.
14. *Яненко Н.Н.* Вопросы модульного анализа и параллельных вычислений в задачах математической физики // Комплексы про-

грамм математической физики. Матер. VI Всесоюзного семинара по комплексам программ мат. физики. Новосибирск, 1980. С. 3–12.
15. *Марчук Г.И.* Методы вычислительной математики. М.: Наука, 1989.
16. *Завьялов Ю.С., Квасов Б.И., Мирошниченко В.Л.* Методы сплайн-функций. М.: Наука, 1980.
17. *Квасов Б. И.* Методы изогеометрической аппроксимации сплайнами. М.: Физматлит, 2006.
18. *Берцун В.Н.* Сплайны сеточных функций. Томск: ТГУ, 2002.
19. *Березин И.С., Жидков Н.П.* Методы вычислений. М.: Наука, 1966. Т.1.
20. *Ильин В.П.* Методы неполной факторизации для решения алгебраических систем. М.: Физматлит, 1995.
21. *Хайрер Э. , Нерсетт С. , Ваннер Г.* Решение обыкновенных дифференциальных уравнений. Нежесткие задачи. М.: Мир, 1990.
22. *Арушанян О. Б. , Залеткин С. Ф.* Численное решение обыкновенных дифференциальных уравнений на Фортране. М.: Изд-во МГУ, 1990.
23. *Фельдман Л. П.* Параллельные алгоритмы численного решения систем линейных обыкновенных дифференциальных уравнений // Математическое моделирование. 2000. Т. 12, №6. С.15–20.
24. *Фельдман Л. П., Назарова И. А.* Эффективность параллельных алгоритмов вложенных методов Рунге – Кутта при моделировании сложных динамических систем // Высокопроизводительные параллельные вычисления на кластерных системах. Материалы второго Международного научно-практического семинара. Нижний Новгород: Изд. ННГУ им Н.И. Лобачевского, 2002.
25. *Оран Э., Борис Дж.* Численное моделирование реагирующих потоков. М.: Мир, 1990.
26. *Вержбицкий В. М.* Численные методы. М.: Высшая школа, 2001.
27. *Высокопроизводительные* вычисления на кластерах / Под ред. А.В. Старченко. Томск: Изд-во Том. ун-та, 2008.
28. *Старченко А.В., Данилкин Е.А., Лаева В.И., Проханов С.А.* Практикум по методам параллельных вычислений. М.: Изд.-во МГУ, 2010.

29. *Копченова Н.В., Марон И.А.* Вычислительная математика в примерах и задачах. М.: Наука, 1972.
30. *Малышкин В.Э., Корнев В.Д.* Параллельное программирование мультикомпьютеров. Новосибирск: Изд-во НГТУ, 2006.
31. *Фадеев Д.К., Фадеева В.Н.* Вычислительные методы линейной алгебры. М.; Л.: Физматгиз, 1969.
32. *Воеводин В.В., Воеводин Вл. В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
33. *Афанасьев К.Е., Домрачев В.Г., Ретинская И.В., Скуратов А.К., Стуколов С.В.* Многопроцессорные системы: построение, развитие, обучение. М.: КУДИЦ-ОБРАЗ, 2005.
34. *Немнюгин С.А*. Средства программирования для многопроцессорных вычислительных систем. СПб.: СПбГУ, 2007.
35. *Гергель В.П.* Теория и практика параллельных вычислений. М.: БИНОМ, 2007.
36. *Воеводин В.В.* Численные методы, параллельные вычисления и информационные технологии. М.: БИНОМ, 2008.
37. *Мышенков В.И., Мышенков Е.В.* Численные методы. Ч. 2: Численное решение обыкновенных дифференциальных уравнений. М.: МГУЛ, 2005.

Tomsk State University

A.V. Starchenko, V.N. Bertsun

Parallel Computing Methods

Textbook

The book includes mathematical foundations of parallel computing and a number of methods for solution of computational mathematics problems on multiprocessor computers with distributed memory: calculating recurrence relations, basic algorithms of linear algebra, direct and iterative methods for solution of linear equations, splines, calculation of definite and multiple integrals, fast Fourier transform, Monte-Carlo method, numerical solution of both ordinary and partial differential equations.

The textbook is addressed to researchers, postgraduate students, undergraduates and bachelors, teachers, who use high-performance computing resources in their scientific and educational work.

The book is recommended by the Council of Classical Universities Teaching Society (Mathematics and Mathematics and Computer Sciences).

Key words: parallel computations, algorithms of computational mathematics, multiprocessor computing systems with distributed memory

Contents

Учебное издание

Старченко Александр Васильевич
Берцун Владимир Николаевич

Методы параллельных вычислений

Учебник

Редактор В.Г. Лихачева
Компьютерная верстка Е.В. Каминская

Подписано в печать 20.12.2012 г.
Формат 60x84 $^{1}/_{16}$. Бумага офсетная №1. Печ. л. 14,6; усл. печ. л. 13,6; уч.-изд. л. 14,2.
Тираж 200 Заказ

ОАО «Издательство ТГУ», 634029, г. Томск, ул. Никитина, 4
ООО «Типография «Иван Федоров»», 634003, г. Томск, Октябрьский взвоз, 1