ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА КЛАСТЕРАХ

Учебное пособие

Под редакцией д-ра физ.-мат. наук А.В. Старченко

Рекомендовано УМС по математике и механике УМО по классическому университетскому образованию РФ в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлениям подготовки «010100 Математика», «010800 Механика и математическое моделирование»

Издательство Томского университета
2008

УДК 519.6
ББК 22.18
В 93

Составители:
Д.А. Беликов, И.В. Говязов, Е.А. Данилкин, В.И. Лаева, С.А. Проханов, А.В. Старченко

В 93 Высокопроизводительные вычисления на кластерах: Учебн. пособие/ Под ред. А.В. Старченко. – Томск: Изд-во Том. ун-та, 2008. – 198 с.
ISBN 978-5-7511-1879-2

В учебном пособии представлены необходимые сведения для работы на многопроцессорной вычислительной системе (кластере) ТГУ: даны основные понятия операционной системы Linux, перечислены правила работы с функциями библиотеки Message Passing Interface, современными компиляторами и технологией OpenMP. Большое внимание уделено практическим вопросам создания параллельных программ для вычисления кратных интегралов, проведения матричных вычислений, решения СЛАУ, численного решения систем ОДУ и уравнений в частных производных. Подробно рассмотрены основные особенности использования математической библиотеки PETSc для параллельных вычислений. Для быстрого освоения и получения практического опыта параллельного программирования в пособии содержится большое количество примеров, необходимых для пользователя, решающего задачи математического моделирования с помощью вычислительной техники.

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

УДК 519.6
ББК 22.18

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

ISBN 978-5-7511-1879-2

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Начиная с момента появления вычислительной техники, человек постоянно стремится улучшить ее производительность, т.е. повысить количество обрабатываемой информации в единицу времени или сократить временные затраты компьютера на выполнение конкретной программы. Осуществление этой цели достигается двумя путями. Первый – непрерывное совершенствование аппаратных элементов вычислительной техники, сокращение связей между ними, повышение скорости их работы. Другой путь повышения производительности – создание суперкомпьютеров за счет увеличения количества исполнителей программы или использования способа обработки информации, отличного от традиционного последовательного. Ясно, что второй путь требует дополнительных затрат при программировании. Но тем не менее он позволяет получать результат быстрее, чем на серийных компьютерах.

Суперкомпьютер – это высокопроизводительная вычислительная система для выполнения сложных расчетов и обработки больших массивов информации.

В настоящее время суперкомпьютерные технологии широко используются в научных исследованиях, при поиске оптимальных технических решений, а также в коммерческой деятельности. С помощью высокопроизводительных вычислительных средств решают задачи следующих научных и научно-технических направлений:

- предсказание погоды, климата и глобальных изменений в атмосфере;
- генетика человека;
- астрономия;
- гидрогазодинамика и теплоперенос;
- прогноз катастрофических явлений в окружающей среде;
- разведка нефти и газа;
- исследование сверхпроводимости;
- распознавание изображений; распознавание и синтез речи;
- автомобиле- и авиастроение;
- прогнозирование в финансовой и экономической областях;
- оптимизация транспортных потоков;
- управление и логистика на крупных и средних предприятиях;
- разработка фармацевтических препаратов;
- обработка больших баз данных;
- защита информации,

а также в стратегических областях:

- управляемый термоядерный синтез, расчет реакторов, расчет поведения плазмы;
- моделирование взрывов и ядерных испытаний;
- разработка военной и авиакосмической техники;
- системы ПВО (распознавание и слежение за большим количеством целей);
- космические разработки и исследования и т.д.

Суперкомпьютеры делятся на:

- *векторно-конвейерные*, особенностью которых является конвейерная обработка данных на специальных функциональных устройствах с использованием векторных команд для работы с целыми массивами данных;
- *массивно-параллельные компьютеры* с распределенной памятью (MPP-машины), построенные путем объединения большого числа (до нескольких тысяч) микропроцессоров с локальной памятью, связанных между собой некоторой высокоскоростной коммуникационной средой;
- *параллельные компьютеры с общей памятью* (SMP-машины), которые отличаются тем, что общая оперативная память в процессе работы вычислительной системы делится между несколькими одинаковыми процессорами, количество которых, как правило, не велико;
- *кластеры*, подобно массивно-параллельным системам, представляют собой многопроцессорные компьютеры, образованные из вычислительных модулей высокой степени готовности, которые связаны между собой системой связи или разделяемыми устройствами внешней памяти. Сейчас часто для кластеров используют в качестве вычислительных узлов обычные серийно выпускаемые компьютеры, а также высокоскоростное сетевое оборудование и специальное программное обеспечение. Из-за малой стоимости комплектующих изделий, возможности постоянного обновления и применения свободно распространяемого программного обеспечения эти системы являются наиболее перспективными с точки зрения получения высокой производительности.

В Томском государственном университете высокопроизводительная многопроцессорная вычислительная система появилась в конце 2000 г. Это кластер **Student**, состоящий из 9 двухпроцессор-

ных узлов с локальной памятью на базе процессоров **Intel Pentium III** с тактовой частотой 650 МГц. Один узел, снабженный 512 Мб оперативной памяти и двумя жесткими дисками объемом 18 Гб, является сервером, на котором происходят компиляция и запуск программ, а также хранятся каталоги пользователей. На остальных узлах установлено по 256 Мб локальной памяти. Они используются наряду с сервером для параллельных расчетов. Вычислительные узлы кластера объединены в локальную компьютерную сеть **Fast Ethernet** с пропускной способностью 100 Мбит/с с коммуникационной топологией «звезда». Пиковая производительность кластера 11,7 Гфлопс, реальная производительность на тесте **Linpack** – 5,6 Гфлопс. На кластере установлено следующее программное обеспечение:

- операционная система **Red Hat Enterprise Linux AS release 4;**
- кластерный пакет **LAM MPI** для организации параллельных вычислений;
- специализированные библиотеки **SCALAPACK**, **Intel MKL**, **FFTW** для решения задач линейной алгебры и применения преобразования Фурье, **SPRNG** для генерации последовательностей псевдослучайных чисел, **FLUENT** и **ANSYS** для инженерных расчетов.

В 2007 г. в ТГУ введен в эксплуатацию вычислительный кластер **СКИФ Cyberia**. Этот суперкомпьютер состоит из 283 вычислительных узлов (один узел – управляющий). На каждом вычислительном узле установлено 4 Гб оперативной памяти, два двухъядерных процессора **Intel Xeon 5150 Woodcrest** с тактовой частотой 2,66 ГГц с поддержкой 64-разрядных расширений, жесткий диск 80 Гб. Управляющий узел с объемом оперативной памяти 8 Гб является сервером, на котором производятся компиляция и запуск программ. Для хранения системных каталогов и каталогов пользователей используется внешняя дисковая система хранения данных объемом 10 Тб.

Вычислительные узлы кластера связаны:

- системной сетью **InfiniPath** для обеспечения одновременной работы всех вычислительных узлов;
- вспомогательной сетью **Gigabit Ethernet** для связывания управляющего и вычислительных узлов;
- сервисной сетью для удаленного независимого включения / выключения питания, консольного доступа к каждому вычислительному узлу.

Пиковая производительность кластера **СКИФ Cyberia** 12 Тфлопс, реальная на тесте **Linpack** – 9,013 Тфлопс. В 2007 г. по своим показателям этот суперкомпьютер занимал 105-е место в списке TOP-500 (29-я редакция) и 1-е место в списке TOP-50 (5-я и 6-я редакции). На кластере **СКИФ Cyberia** установлено следующее программное обеспечение:

- операционная система **Linux SUSE Enterprise Server 10.0**;
- операционная система **Microsoft Windows Compute Cluster Server 2003**;
- поддержка стандартов параллельного программирования **MPI** и **OpenMP**;
- средства разработки приложений – высокопроизводительные оптимизирующие компиляторы с языков **C/C++/Fortran** (**Intel 9.1, PGI 7.0**);
- средства параллельной отладки и трассировки приложений (**Intel Vtune**, **TotalView 8.3**);
- системы управления заданиями и мониторинга для суперкомпьютера (**Ganglia**);
- специализированное программное обеспечение – пакеты для научных и инженерных расчетов **Fluent** и **ANSYS**, моделирующие системы для исследований погоды и климата **MM5** и **WRF**, системы для решения задач охраны окружающей среды **CAMx** и **CMAQ**; программа для докинга низкомолекулярных лигандов в белковые рецепторы **AUTODOCK** и комплекс программ для визуализации и анализа молекулярной структуры **MGLTools**, квантовохимический комплекс **GAMESS**;
- библиотеки для проведения параллельных вычислений **PETSc**, **FFTW**, **MKL**, **SPRNG**.

Прежде чем приступить к написанию параллельной программы для исследовательских целей, необходимо выяснить следующие вопросы:

1) Будет ли созданная параллельная программа работать быстрее, чем ее последовательные варианты?
2) Соизмерим ли полученный выигрыш во времени, который дает параллельная программа, с затратами на программирование?

Частичные ответы на эти вопросы могут быть получены на основе закона Амдаля. Он устанавливает связь между ускорением S_p

работы параллельной программы по сравнению с последовательной, числом процессоров p и долей вычислительных операций параллельной программы $\alpha\,(0 \le \alpha \le 1)$, которые выполняются сугубо последовательно:

$$S_p = \frac{1}{\alpha + (1-\alpha)/p}.$$

Обычно ускорение S_p определяется как отношение процессорного времени T_1 на выполнение последовательной программы на одном процессоре ко времени T_p выполнения вычислений параллельной программой на p-процессорной машине. При этом предполагается, что обе программы реализуют один и тот же алгоритм.

Если в параллельной программе доля последовательных операций (т.е. операций, которые обязательно должны выполняться одна за другой, а не одновременно) не может быть уменьшена более чем наполовину $(\alpha > 0,5)$, то ускорение параллельной программы будет

$$S_p = \frac{2}{1+1/p} < 2,$$

невзирая на количество процессоров, используемых в расчетах.

Заметим также, что формулировка закона Амдаля не учитывает временных затрат на обеспечение обмена информацией между вычислительными узлами. Поскольку эти затраты (обусловленные необходимостью обеспечения коммуникационных обменов, предотвращения конфликтов памяти, синхронизации) лишь увеличивают общее время работы параллельной программы, то реальное ускорение S_p будет еще ниже.

Поэтому прежде чем приступить к распараллеливанию вычислительной процедуры и проектированию программы для многопроцессорной машины, необходимо путем простого пересчета оценить время, которое будет затрачено на осуществление арифметических операций и межпроцессорные обмены при выполнении параллельного алгоритма. После этого следует соотнести полученный результат с итогом аналогичных расчетов для последовательного алгоритма и сделать вывод о перспективности планируемых действий.

1 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ И ПОРЯДОК РАБОТЫ НА ВЫЧИСЛИТЕЛЬНОМ КЛАСТЕРЕ ТГУ СКИФ Cyberia

1.1 Программное обеспечение

Вычислительный кластер ТГУ СКИФ Cyberia работает под управлением операционных систем – **Suse Linux Enterprise Server 10** и **Microsoft Windows Compute Cluster Server 2003**.

Suse Linux Enterprise Server 10 (**SLES**) – один из дистрибутивов операционной системы Linux, представляющий собой масштабируемую, высокопроизводительную платформу безопасных корпоративных вычислений, реализующую все преимущества **Linux** и **OpenSource**.

Microsoft Windows Compute Cluster Server 2003 (**CCS**) является интегрированной, стандартной кластерной платформой, основанной на высокопроизводительной системе **Windows Server 2003 x64**. Это решение реализует возможности высокопроизводительных вычислений в обычном рабочем окружении **Windows**.

На кластере установлены компиляторы **gcc 4.1.0** и **Intel C/C++/Fortran Compilers**.

GCC (**GNU Compiler Collection**) – набор свободно доступных компиляторов для различных языков программирования (**C/C++/Fortran** и др.). Он используется как стандартный компилятор для свободных Unix-подобных операционных систем.

Intel C/C++/Fortran Compilers – компиляторы нового поколения, позволяющие обеспечить высокую производительность на процессорах **Intel**.

Компиляторы **Intel** дополнены эффективными средствами разработки многопоточного кода, поддерживающими стандарт **OpenMP** и технологию автораспараллеливания кода. Например, библиотека **Intel Math Kernel Library 9.0** (**Intel MKL**) предоставляет набор оптимизированных математических функций, позволяющих создавать максимально быстродействующие инженерные, научные и финансовые приложения для процессоров **Intel**. Функциональные области библиотеки включают линейные алгебраические функции **LAPACK** и **BLAS**, быстрые преобразования Фурье и трансцендентные векторные функции (библиотека векторной математики VML).

На кластере используется параллельная среда **QLogic MPI**. **Qlogic MPI** – библиотека **MPI** для работы с высокопроизводительной сетью **Infiniband**.

Установлен анализатор производительности **Intel VTune**. Анализатор **Intel VTune** собирает и отображает данные о производительности программы, позволяет определить и локализовать «узкие места» программного кода.

MM5 – негидростатическая мезомасштабная модель для исследования региональной погоды над ограниченной поверхностью со сложным рельефом.

Также на кластере СКИФ Cyberia для выполнения инженерных расчетов используются специализированные пакеты **ANSYS**, **FLUENT**.

ANSYS – многоцелевой конечно-элементный пакет для проведения анализа в широкой области инженерных дисциплин (акустика, прочность, деформация, теплофизика, динамика жидкостей и газов и электромагнетизм и др.).

FLUENT – современный программный комплекс, предназначенный для решения задач механики жидкости и газа. Пакеты адаптированы для работы на кластерной системе.

Установленная на кластере библиотека **PETSc** (**Portable, Extensible Toolkit for Scientific Computation**) используется для численного решения дифференциальных уравнений в частых производных.

1.2 Порядок работы на кластере СКИФ Cyberia

Для входа на кластер с персонального компьютера, имеющего выход в Интернет, необходимо использовать ssh-клиент.

Пользователям операционной системы **Microsoft Windows** рекомендуется использовать программу ***putty***.
(http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html).

Чтобы установить соединение с кластером при помощи **putty**, необходимо:

1. В поле **Host Name** ввести **cyberia.tsu.ru**, port 22.
2. Connection type: **SSH**.
3. В **Category — Window — Translation — Character set translation on received data** выбрать **UTF-8**.
4. В **SSH — Preferred SSH protocol version** должно быть указано 2.

5. В поле **Saved Sessions** ввести любое удобное название. Нажать кнопку «**Save**». Под этим именем будут сохранены настройки.

6. Теперь для входа на кластер достаточно загрузить сохраненные настройки, выбрать их в меню **Saved sessions** и нажать «**Load**».

7. Нажать «**Open**».

Сеанс работы с операционной системой начинается с ввода идентификатора пользователя (**username**) и его пароля (**password**). Приглашением к вводу идентификатора является фраза «***login as***». Пароль при вводе не отображается.

Пользователи Unix-подобных операционных систем также могут воспользоваться putty или использовать стандартный консольный ssh-клиент.

Чтобы установить соединение с кластером при помощи стандартного ssh-клиента, необходимо:

1. В командной строке выполнить команду:

ssh -l username cyberia.tsu.ru или **ssh username@cyberia.tsu.ru**

2. Ввести пароль (**password**).

Для обмена файлами с кластером пользователям **ОС Microsoft Windows** рекомендуется применять программу **WinSCP** (**http://winscp.net/eng/docs/lang:ru**) или консольный клиент **psftp** из пакета **putty**. **WinSCP** выполняет все основные операции с файлами, такие как загрузка и выгрузка файлов. Он также позволяет переименовывать файлы и папки, создавать папки, изменять свойства файлов и папок.

Для передачи файлов через **WinSCP** необходимо:

1. В поле **Host Name** указать **cyberia.tsu.ru**, port 22.
2. Указать **User name**.
3. File Protocol: **SFTP**.
4. Сохранить настройки.
5. Нажать «**login**», ввести пароль.

Работа с **WinSCP** во многом похожа на работу **Norton Commander** (**Far**). Также существует возможность использовать **WinSCP Far plugin**. (**http://winscp.net/eng/docs/fm_plugins**).

Для передачи файлов с помощью **psftp** необходимо:

1. Запустить **psftp**.
2. Вызвать команду **open username@cyberia.tsu.ru**.

3. Ввести пароль.

Основные команды **psftp**:

ls – показывает список файлов и подкаталогов;

get filename – «скачивает» файл filename на ваш компьютер;

put filename – загружает файл filename из локального каталога на удаленный компьютер;

cd dir – устанавливает текущим каталогом удаленный каталог с именем *dir*.

Дополнительную справку можно получить командами «*help*» и «*help имя_команды*».

Под Unix-подобными операционными системами можно пользоваться программами **sftp**, **scp** или **Midnight Commander** (пункт меню «shell link»).

1.3 Операционная система Linux

Операционная система (ОС) – *это комплекс программ, обеспечивающих управление ресурсами компьютера и процессами, выполняющимися на компьютере, а также взаимодействие пользователей и компьютера.*

Linux — это свободная **UNIX**-подобная операционная система. На кластере установлена операционная система **Suse Linux Enterprise Server 10** (SLES).

1.3.1 Интерфейс ОС Linux

В операционной системе **Linux** при наборе команд, имен файлов и каталогов различаются заглавные и прописные буквы. Процесс общения с **ОС Linux** осуществляется в основном при помощи консольного терминала (при помощи клавиатуры и монитора). После входа в систему на экране можно увидеть один из двух элементов интерфейса **Linux**: командную строку или графическую оболочку.

Командная строка – традиционный способ общения в **Linux**. Когда **Linux** ожидает команду, на экране высвечивается приглашение – «**$**». Чтобы дать команду системе, нужно в командной строке набрать имя команды, а затем – ее опции и аргументы (могут отсутствовать), разделенные пробелами. После набора ввод команды осуществляется нажатием клавиши ENTER. Команды **Linux** имеют структуру

command option(s) argument(s)

Опции обычно сопровождаются знаком «минус» и модифицируют поведение команды. Аргументы – это обычно имена каталогов или файлов. Следует помнить, что строчные и заглавные буквы в **Linux** обрабатываются по-разному, поэтому, например, команды **man** и **Man** воспринимаются ОС как различные. Это распространяется и на имена файлов.

Заметим, что каталог пользователя находится в папке **/home/user_name** корневого каталога системы.

1.3.2 Некоторые команды Linux

Ниже приводятся некоторые наиболее употребляемые команды **Linux**. Чтобы получить более полную информацию по любой отдельной команде ***command***, нужно ввести

man command

Выход из описания команды производится при нажатии клавиши «***q***».

1.3.3 Работа с каталогами

pwd – показывает название текущей директории;

cd dir – устанавливает текущим каталогом каталог с именем *dir,* вызов команды *cd* без параметров возвращает в домашний каталог /home/username ($HOME);

mkdir subdir – создает новый подкаталог с именем *subdir*;

rmdir subdir – удаляет пустой подкаталог с именем *subdir;*

ls – показывает список файлов и подкаталогов текущей директории;

ls dir – показывает список файлов и подкаталогов каталога *dir*;

ls –A – показывает все файлы, в том числе и скрытые;

ls –l – показывает атрибуты (владелец, разрешение на доступ, размер файла и время последней модификации);

mv oldname newname – изменяет имя подкаталога или перемещает его;

cp -R dirname destination – копирует подкаталог dirname в другое место *destination*.

1.3.4 Работа с файлами

file filename(s) – определяет тип файла (например, ASCII, JPEG image data и др.);

cat filename(s) – показывает содержание файлов (используется только для текстовых файлов!);

more filename(s) – действует так же, как и *cat*, но позволяет листать страницы;

less filename(s) – улучшенный вариант команды *more*;

head filename – показывает первые десять строк файла *filename*;

tail filename – показывает последние десять строк файла *filename*;

wc filename(s) – показывает число строк, слов и байт для указанного файла;

rm filename(s) – уничтожает файлы или директории, для рекурсивного удаления следует использовать *rm* с ключом *-rf*.

cp filename newname – создает копии файлов с новыми именами;

cp filename(s) dir – копирует один или более файлов в другой каталог;

mv oldname newname – изменяет имя файла или каталога;

mv filename(s) dir – перемещает один или более файлов в другой каталог;

find dir -name filename – пытается локализовать файл (подкаталог) *filename* рекурсивно в подкаталоге *dir*.

1.3.5 Другие полезные команды

passwd – изменяет пароль пользователя системы **Linux**; требует подтверждения старого;

who – показывает, кто в настоящее время работает в сети;

finger – дает более подробную информацию о пользователях сети;

write – позволяет послать сообщение пользователю, работающему в сети в данное время;

top – отображает информацию о процессах, использующих процессоры узла;

ps –U user_name – показывает номера процессов (pid), инициированных пользователем *user_name*;

kill xxxxx – досрочно завершает работу процесса с номером *xxxxx*;

killall proc_name – досрочно завершает работу процесса proc_name;

date – отображает дату и время;

cal – показывает календарь;

exit – выйти из терминала;

clear – очистить окно терминала;

du dir – показывает занятое место в директории *dir*.

Для облегчения взаимодействия с ОС были предложены программы-оболочки, которые делают наглядным и простым выполнение следующих базовых операций над файлами и каталогами:

- перемещение по каталогам и подкаталогам;
- создание и удаление подкаталога;
- просмотр, редактирование файла;
- копирование, перенос, переименование, удаление файлов;
- запуск программ и др.

*В **ОС Linux** к таким программам-оболочкам относится **Midnight Commander**, который в основных чертах аналогичен известным оболочкам операционных систем **DOS** и **Windows** – **Norton Commander** и **FAR**. Запуск **Midnight Commander** осуществляется выполнением команды **mc**. Выход производится после последовательного нажатия клавиш **F10** и **Enter**.*

1.3.6 Редактирование файлов

Текстовый редактор необходим для изменения содержания текстового файла или подготовки и редактирования текста программы на каком-либо языке программирования. В операционной системе **Linux** кластера ТГУ СКИФ Cyberia используются редакторы ***vim***, ***mcedit***.

Редактор ***vim*** – простой, но очень мощный, однако неопытным пользователям не рекомендуется его применять. Выход из редактора без сохранения изменений редактируемого файла – ***:qa!***. Подробнее о его работе можно узнать, выполнив команду

vimtutor

Встроенный в **Midnight Commander** редактор ***mcedit*** гораздо проще в работе.

Запуск этих редакторов в командной строке производится следующим образом:

vi(m) filename
mcedit filename

Если же используется программа-оболочка (**Midnight Commander**), то нужно установить курсор на ***filename*** и нажать клавишу **F4**. Загрузится редактор, настроенный на действие этой опции.

1.4 Компилирование последовательных программ (Fortran/C/C++)

После подготовки текста программы с помощью текстового редактора необходимо получить исполняемый файл, запуск которого позволит провести расчеты по написанной программе.

На вычислительном кластере установлены компиляторы **Intel** (**icc**, **ifort**) и компиляторы **GNU** (**gcc**, **gfortran**). Рекомендуется пользоваться компиляторами **Intel**.

В случае, если была подготовлена программа на языке **FORTRAN**, соответствующий текстовый файл должен иметь расширение ***.f*** или ***.for*** (например, ***myprog.f***). Для программы, написанной в соответствии со стандартом **FORTRAN 90,** - *.f90*. Чтобы произвести его компиляцию, нужно ввести команду

ifort –O myprog.f –o myprog.exe

Опция **–O** означает, что будет создан рабочий модуль, оптимизированный с точки зрения скорости выполнения программы, если компилятор не обнаружит ошибки. Подчеркнутая часть команды говорит о том, что рабочий модуль будет иметь имя ***myprog.out***. Если же эта часть будет опущена, то имя рабочей программы будет ***a.out***.

Для компиляции программ, написанных на языке программирования **C** (**C++**), используется программа ***icc***. Более подробно о ***ifort*** и ***icc*** можно узнать, выполнив команды

man ifort
man icc

Для запуска рабочего модуля последовательной программы необходимо использовать установленную систему очередей (см. п.1.7).

1.5 Создание параллельной программы с использованием MPI

Библиотека **MPI** (**Message Passing Interface**) была разработана для создания параллельных программ. Реализация MPI-программ позволяет наиболее оптимально использовать кластерный вычислительный ресурс. На вычислительном кластере установлена библиотека MPI, оптимизированная для работы с высокопроизводительной средой передачи данных **InfiniPath**, входящей в состав вычислительного комплекса.

Кроме самой библиотеки на кластере установлены «привязки» к языкам программирования **C**, **C++** и **Fortran**. Для облегчения компиляции и сборки программ рекомендуется использовать следующие утилиты:

mpicc – для программ, написанных на языке программирования **C**;

mpixx (mpiCC) – для программ, написанных на языке программирования **C++**;

mpif77 и mpif90 – для программ, написанных на языке программирования **Fortran**.

Синтаксис данных утилит во многом похож на синтаксис компиляторов ***icc*** и ***ifort***, более полная информация о синтаксисе доступна по команде ***man имя_утилиты***.

Например, для компиляции программы **fisrt.f90** можно выполнить команду

mpif90 -O first.f90 -o first.exe

1.6 Запуск параллельной MPI-программы

Запуск MPI-приложения на вычислительном кластере возможен только через систему пакетной обработки заданий (см. п. 1.7).

Для упрощения запуска и постановки в очередь параллельной программы предусмотрен специальный скрипт ***mpirun***.

Например,

mpirun -np 20 ./first.exe

запустит параллельную программу ***first.exe*** на 20 процессорах, т.е. на 5 узлах (каждый узел имеет 2 двухъядерных процессора). Стоит обратить внимание, что для запуска исполняемого модуля, находящегося в текущей директории (**$pwd**), необходимо явно указать путь «**./**»

1.7 Работа с системой пакетной обработки задач

На вычислительном кластере установлена система пакетной обработки заданий ***torque***. Данная система предназначена для контроля вычислительных ресурсов и выделения их под задачи пользователя.

Скрипт ***mpirun*** помещает задачу в очередь, но при таком подходе на каждом узле будет использоваться максимальное количество процессоров/ядер, т.е. задача будет запущена 4 раза на каждом вычислительном узле. В некоторых случаях, например при использовании **OpenMP**, **MPI+OpenMP** или для запуска последовательных программ, целесообразно использовать систему пакетной обработки заданий.

1.7.1 Основные команды

qstat – утилита для просмотра состояния очереди заданий. По команде «***qstat -f***» на экране выводится полный список задач пользователя с параметрами.

qsub – утилита для установки задачи в очередь на выполнение. По данной команде в очередь на выполнение будет поставлен сценарий оболочки, из которого происходит запуск самой вычислительной задачи. При постановке задачи на выполнение пользователь указывает требуемое количество ресурсов, а также дополнительные параметры, такие, как рабочая директория, имена файлов вывода стандартных потоков **stdout** и **stderr**, имена переменных для передачи в качестве системного окружения задачи и т.д. Полное описание ресурсов задачи можно получить по команде **man pbs_resources**, а описание атрибутов задачи – по команде **man pbs_job_attributes**.

Пример:

qsub script.sh

Синтаксис ***script.sh***

#PBS -o $DIR/stdout.log
Определяет имя файла, в который будет перенаправлен стандартный поток **stdout.**

#PBS -e $DIR/stderr.log
Определяет имя файла, в который будет перенаправлен стандартный поток **stderr.**

#PBS -l nodes=8:ppn=2:cpp=1
Определяет, какое количество узлов и процессоров на них необходимо задействовать:
nodes – количество узлов;
ppn – число процессоров на узле;
cpp – число процессов на процессоре.

#PBS -l walltime=20:00:00
Определяет максимальное время счета задания.

#PBS -l mem=4000mb
Определяет количество необходимой оперативной памяти.

cat $PBS_NODEFILE | grep -v master | sort | uniq -c | awk '{printf "%s:%s\n", $2, $1}' > $PBS_O_WORKDIR/temp.tmp
Составляет список узлов в необходимом формате, на которых будет запущена задача, и записывает их в файл **temp.tmp.**

cd $PBS_O_WORKDIR
/usr/bin/mpirun.ipath -m temp.tmp –np 16 ./a.out
Запускает на узлах указанных в файле **temp.tmp** 16 копий параллельной программы.

Пример скрипта:

```
#PBS -o $DIR/stdout.log
#PBS -e $DIR/stderr.log
#PBS -l nodes=50:ppn=2
#PBS -l walltime=20:00:00
#PBS -l mem=4000mb
cat $PBS_NODEFILE | grep -v master | sort | uniq -c   | awk
'{printf "%s:%s\n", $2, $1}' > $PBS_O_WORKDIR/script.mf
cd $PBS_O_WORKDIR
/usr/bin/mpirun.ipath -m script.mf -np 100 ./a.out
```

Здесь будет запущена параллельная программа **a.out** на 50 узлах, с каждого узла будет использоваться 2 процессора. Файл вывода стандартного потока **stdout** — **stdout.log**, стандартного потока **stderr** — **stderr.log**. **$DIR** содержит путь к файлам **stdout.log** и **stderr.log**, например, может принимать значение **/home/user_name**. Под задачу отведено 20 часов. Необходимое количество памяти 1000 мегабайт.

Для запуска последовательной программы ***first.exe*** можно использовать следующий скрипт:

```
#PBS -o $DIR/stdout.log
#PBS -e $DIR/stderr.log
#PBS -l walltime=10:00
#PBS -l mem=100mb
./first.exe
```

При запуске программы через команду **qsub** заданию присваивается уникальный целочисленный идентификатор.

```
qsub script.sh
Starting MPI job under Torque resource manager...
20213.master.cyberia.tsu.ru
```

Здесь идентификатор задания 20213.

qdel – утилита для удаления задачи. В случае, если задача уже запущена, процесс ее работы будет прерван. Синтаксис данной утилиты следующий:

qdel [-W время задержки] идентификаторы задачи.

Выполнение такой команды удалит задачи с заданными идентификаторами через указанное время.

Более подробное описание синтаксиса команд можно получить, выполнив команду

man имя_утилиты.

2 СОВРЕМЕННЫЕ КОМПИЛЯТОРЫ ПРОГРАММ. КОМПИЛЯТОР INTEL COMPILER 9.1

2.1 Общие сведения

Компилятор **Intel** позволяет достичь высочайшей производительности **Windows** и **UNIX** приложений на 32- и 64-разрядных платформах **Intel**, включая системы на базе процессоров **Pentium M**, **Pentium 4** с технологией **Hyper-Threading**, **Xeon**, **Itanium** и **Itanium 2**. Компилятор предлагает мощные средства оптимизации кода, встроенную поддержку многопоточных приложений и инструменты для реализации технологии **Hyper-Threading**. Он поддерживает популярные средства разработки и индустриальные стандарты языка **Fortran**. Вспомогательные инструменты для тестового покрытия кода и установки приоритетов тестирования помогают существенно сократить период отладки и тестирования приложений.

Компиляторы **Intel** вызываются с помощью команд **icc** (**C** или **C++**), **icpc** (**C++**) и **ifort** (**Фортран 77/90**). Команды **mpicc**, **mpiCC** и **mpif77** для компиляции и сборки MPI-программ также настроены на использование компиляторов **Intel**.

По умолчанию файлы с расширением **.cpp** и **.cxx** считаются исходными текстами на языке **C++**, файлы с расширением **.c** – исходными текстами на языке **C**, а компилятор **icpc** также компилирует файлы **.c** как исходные тексты на **C++**.

Файлы с расширениями **.f**, **.ftn** и **.for** распознаются как исходные тексты на языке **Фортран** с фиксированной формой записи, а файлы **.fpp** и **.F** дополнительно пропускаются через препроцессор языка Фортран. Файлы с расширением **.f90** считаются исходными текстами **Фортран 90/95** со свободной формой записи. Явным образом можно задать фиксированную или свободную форму записи Фортран-программ с помощью опций **-FI** и **-FR** соответственно.

Файлы с расширением **.s** распознаются как код на языке ассемблера для **IA-32**.

Компиляторы Intel характеризуются следующими возможностями:

- значительной оптимизацией исходной программы на высоком уровне, т.е. прежде всего выполняют различные преобра-

зования циклов, что с большим или меньшим успехом делают почти все компиляторы;

- оптимизацией вычислений с плавающей точкой путем максимального использования команд, реализованных на аппаратном уровне;
- межпроцедурной оптимизацией, т.е. глобальной оптимизацией всей программы в отличие от обычной оптимизации, которая затрагивает только код конкретных функций;
- оптимизацией на базе профилей, т.е. возможностью прогнать программу в тестовом режиме, собрать данные о времени прохождения тех или иных фрагментов кода внутри часто вызываемых функций, а затем использовать эти данные для оптимизации;
- поддержкой системы команд SSE в процессорах Pentium III;
- автоматической векторизацией, т.е. использованием команд SSE и SSE2, вставляемых автоматически компилятором;
- поддержкой OpenMP для программирования на SMP-системах;
- предвыборкой данных – использованием команд предварительной загрузки из памяти в кэш данных, которые понадобятся через некоторое время;
- «диспетчеризацией» кода для различных процессоров, т.е. возможностью генерации кода для различных процессоров в одном исполняемом файле, что позволяет использовать преимущества новейших процессоров для достижения на них наибольшей производительности.

Согласно результатам выполнения тестов SPEC CPU2000, опубликованным на сервере **http://www.ixbt.com**, компиляторы **Intel** версии 6.0 практически везде оказались лучше по сравнению с компиляторами **gcc** версий 2.95.3, 2.96 и 3.1 и **PGI** версии 4.0.2. Эти тесты проводились в 2002 г. на компьютере с процессором **Pentium 4**/1.7 ГГц и **ОС RedHat Linux 7.3**.

Согласно результатам тестов, проведенных компанией **Polyhedron**, компилятор **Intel Fortran** версии 7.0 почти везде оказался лучше по сравнению с другими компиляторами **Fortran 77** для **Linux** (**Absoft**, **GNU**, **Lahey**, **NAG**, **NAS**, **PGI**). Только в некоторых тестах компилятор **Intel** незначительно проигрывает компиля-

торам **Absoft**, **NAG** и **Lahey**. Эти тесты были проведены на компьютере с процессором **Pentium 4**/1.8 ГГц и **ОС Mandrake Linux 8.1**.

Компиляторы **Intel** версии 9.1 также обгоняют по производительности компиляторы **gcc** и показывают производительность, сравнимую с **Absoft**, **PathScale** и **PGI**.

2.2 Совместное использование модулей на Фортране и Си

Чтобы совместно использовать модули, написанные на языках Фортран и Си, нужно согласовать наименование процедур в объектных модулях, передачу параметров, а также доступ к глобальным переменным, если такие есть.

По умолчанию компилятор **Intel Fortran** переводит имена процедур в нижний регистр и добавляет в конец имени процедуры знак подчеркивания. Компилятор Си никогда не изменяет имена функций. Таким образом, если мы хотим из модуля на Фортране вызвать функцию или процедуру FNNAME, реализованную на Си, то в модуле на Си она должна именоваться fnname_.

Компилятор Фортрана поддерживает опцию **-nus [имя файла]**, которая позволяет отключать добавление знаков подчеркивания к внутренним именам процедур. Если задано имя файла, то это производится только для имен процедур, перечисленных в заданном файле.

По умолчанию на Фортране параметры передаются по ссылке, а на Си – всегда по значению. Таким образом, при вызове Фортран-процедуры из модуля на Си необходимо в качестве параметров передавать указатели на соответствующие переменные, содержащие значения фактических параметров. При написании на Си функции, которую надо будет вызывать из модуля на Фортране, требуется описывать формальные параметры как указатели соответствующих типов.

В модулях на Си возможно использование COMMON-блоков, определенных внутри модулей на Фортране (подробнее об этом см. **Intel Fortran Compiler User's Guide**, глава **Mixing C and Fortran**).

2.3 Основные опции компиляторов

Наиболее интересными, конечно же, являются опции оптимизации кода. Большинство опций совпадают для **Intel** компиляторов **C++** и Fortran.

Уровни оптимизации	
Опция	**Описание**
-O0	Отключает оптимизацию
-O1 или -O2	Базовая оптимизация, ориентированная на скорость работы. Отключается инлайн-вставка библиотечных функций. Для компилятора **C++** эти опции дают одинаковую оптимизацию, для компилятора Фортрана опция -O2 предпочтительнее, т.к. включает еще раскрутку циклов. Раскрутка циклов заключается в дублировании тела цикла и сокращении числа итераций. Кроме того, необходимо позаботиться, чтобы все вхождения переменной цикла в тело были заменены выражениями, вычисляющими требуемые значения с учетом измененных значений переменной цикла
-O3	Более мощная оптимизация, включая преобразования циклов, предвыборку данных, использование **OpenMP**. На некоторых программах может не гарантироваться повышенная производительность по сравнению с **-O2**. Имеет смысл использовать вместе с опциями векторизации **-xK** и **-xW**
-unroll[n]	Включает раскрутку циклов до **n** раз
Оптимизации под конкретный процессор	
Опция	**Описание**
-tpp6	Оптимизация для процессоров Penitum Pro, Pentium II и Pentium III
-tpp7	Оптимизация для процессоров Penitum 4 (эта опция включена по умолчанию для компилятора на IA-32)
-xM	Генерация кода с использованием расширений MMX, специфических для процессоров Pentium MMX, Pentium II и более поздних
-xK	Генерация кода с использованием расширений SSE, специфических для процессоров Pentium III
-xW	Генерация кода с использованием расширений SSE2, специфических для процессоров Pentium 4
Межпроцедурная оптимизация	
-ip	Включается межпроцедурная оптимизация внутри одного файла. Если при этом указать опцию **-ip_no_inlining**, то отключаются инлайн-вставки функций
-ipo	Включается межпроцедурная оптимизация между различными файлами
Оптимизации с использованием профилей	
-prof_gen	Генерируется «профилировочный» код, который будет использован для профилировки, т.е. сбора данных о частоте прохождения тех или иных участков программы
-prof_use	Производится оптимизация на основе данных, полученных на этапе профилировки. Имеет смысл использовать вместе с опцией межпроцедурной оптимизации **-ipo**
Распараллеливание для SMP-систем	
-openmp	Включается поддержка стандарта OpenMP 2.0
-parallel	Включается автоматическое распараллеливание циклов

Более подробное описание можно посмотреть в руководстве пользователя **Intel Fortran Compiler User's Guide.**

В табл. 2.1 представлено время центрального процессора в секундах, затраченное на перемножение матриц размерностью 1000x1000, 2000x2000, 4000x4000, 8000x8000 элементов для различных уровней оптимизации. Анализируя данные, следует отметить, что правильный выбор параметров оптимизации позволяет существенно сократить время решения задачи.

Таблица 2.1 Время ЦП, в с, при умножении матриц различной размерности

	1000	2000	4000	8000
O0	5,89	47,57	380,32	2854,6
O2	2,28	20,25	163,48	1312,3
O3	2,28	20,28	163,27	1312,6

Время счета Fortran-программ, откомпилированных с использованием опций -O2 и -O3, практически не отличается, однако компиляция с ключом -O3 происходит гораздо дольше.

2.4 Технологии OpenMP

OpenMP (**Open specifications for Multi-Processing**) – стандарт для написания параллельных программ для многопроцессорных вычислительных систем с общей оперативной памятью. Программа представляется как набор потоков (нитей), объединённых общей памятью, где проблема синхронизации решается введением критических секций и мониторов. Разработкой стандарта занимается организация **OpenMP ARB** (**ARchitecture Board**), в которую вошли представители крупнейших компаний-разработчиков SMP-архитектур (***Symmetric Multiprocessing***) и программного обеспечения. Спецификации для языков **Fortran** и **C/C++** появились соответственно в октябре 1997 г. и октябре 1998 г. **OpenMP** – это набор специальных директив компилятору, библиотечных функций и переменных окружения. Наиболее оригинальны директивы компилятору, которые используются для обозначения областей в коде с возможностью параллельного выполнения. Компилятор, поддерживающий **OpenMP**, преобразует исходный код и вставляет соответствующие вызовы функций для параллельного выполнения этих областей кода.

За счет идеи «частичного распараллеливания» **OpenMP** идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP-директивы. Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости одновременно поддерживать последовательную и параллельную версии. Директивы **OpenMP** просто игнорируются последовательным компилятором, а для вызова процедур **OpenMP** могут быть подставлены заглушки (**stubs**), текст которых приведен в спецификациях.

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы pragma и функции исполняющей среды **OpenMP**. Директивы pragma, как правило, указывают компилятору, как реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с фразы ***!$OMP***. Как и любые другие директивы pragma, они игнорируются компилятором, не поддерживающим конкретную технологию – в данном случае **OpenMP**. Каждая директива может иметь несколько дополнительных атрибутов. Отдельно специфицируются атрибуты для назначения классов переменных, которые могут быть атрибутами различных директив.

Функции **OpenMP** служат в основном для изменения и получения параметров окружения. Кроме того, **OpenMP** включает API-функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции **OpenMP** библиотеки периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл **omp.h**. Если же используется в приложении только OpenMP-директивы pragma, включать этот файл не требуется.

2.4.1 Основные директивы OpenMP

!$OMP PARALLEL [*атрибут* [[,] *атрибут*] …]
блок операторов программы
!$OMP END PARALLEL

Определяет *параллельную область* программы. При входе в эту область порождаются новые (N-1) нити, образуется «команда» из N

нитей, а порождающая нить получает номер 0 и становится основной нитью команды (так называемой «master thread»). При выходе из *параллельной области* основная нить дожидается завершения остальных нитей и продолжает выполнение программы в одном экземпляре. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы). Атрибут – это необязательный модификатор директивы, влияющий на ее выполнение. Атрибуты могут быть следующими:

- **PRIVATE** (*список*)

Каждая нить имеет свою собственную неинициализированную локальную копию переменных и общих блоков, перечисленных в *списке*.

- **SHARED** (*список*)

Этот атрибут определяет, что переменные, перечисленные в *списке*, являются доступными для всех нитей. Если переменная определяется как **SHARED**, то тем самым утверждается, что все нити могут работать только с одной копией переменной.

- **DEFAULT** (*атрибут*)

Атрибут может иметь одно из следующих значений:

PRIVATE
SHARED
NONE

Когда в директиве **PARALLEL** используется атрибут **DEFAULT**, то тем самым объявляется, что все переменные в блоке операторов *параллельной области* программы должны быть **PRIVATE**, **SHARED** или **NONE**.

- **FIRSTPRIVATE** (*список*)

Этот атрибут используется подобно **PRIVATE** за исключением того, что каждая нить получает инициализированные копии переменных, перечисленных в *списке*.

- **REDUCTION** ({*оператор*|*встроенная функция*}:*список*)

Здесь *оператор*|*встроенная функция* могут принимать одно из следующих значений (начальное значение переменной из *списка* приведено в скобках):

+ (0)
* (1)
- (0)
.AND.(.TRUE.)
.OR. (.FALSE.)

.NEQV. (.FALSE.)
MAX (наименьшее «компьютерное» число)
MIN (наибольшее «компьютерное» число)
IAND (все биты равны единице)
IOR (0)
IEOR (0)

Когда установлен атрибут **REDUCTION**, окончательные локальные значения (для каждой нити) для каждой переменной из списка получаются с учетом выбранного оператора или встроенной функции редукции. Это значение затем комбинируется со значениями локальных переменных из *списка*, полученных с остальных нитей перед завершением параллельной области. Внутри *параллельной области* переменные из *списка* имеют атрибут **PRIVATE** и инициализируются, как указано выше.

- **IF** (*логическое_выражение*)

Если *логическое_выражение* имеет значение «истина», то блок выполняется параллельно, если – «ложь», то последовательно.

- **COPYIN** (*список имен общих блоков*)

Атрибут **COPYIN** применяется только к общим блокам, которые объявлены как **THREADPRIVATE** (локальные). В параллельной области **COPYIN** определяет, какие данные (общие блоки) из главной (порождающей) нити должны быть скопированы на другие нити в начале *параллельной области*.

!$OMP DO [*атрибут*[[,] *атрибут*] …]
do_цикл
!$OMP END DO [NOWAIT]

Директива **DO** определяет, что итерации непосредственно следующего *цикла DO* должны быть выполнены параллельно. Директива **DO** должна обязательно располагаться в *параллельной области* программы, поскольку она сама не создает нитей.

Атрибуты могут быть следующими:

- **PRIVATE** (*список*)
- **FIRSTPRIVATE** (*список*)
- **REDUCTION** ({*оператор*|*встроенная функция*}:*список*)
- **SCHEDULE** (*тип* [,*количество*]))

Атрибут **SHEDULE** задает, как итерации цикла DO будут распределяться между нитями. *Количество* оценивается из контекста конструкции директивы **DO**.

- **SCHEDULE (STATIC, m)** – итерации делятся на блоки размером, определенным в **m**. Блоки назначаются нитям циклически.
- **SCHEDULE (DYNAMIC, m)** – итерации делятся на блоки размером, определенным в **m**. Как только нить закончит выполнение блока, она динамически получает следующий блок итераций для выполнения.
- **SCHEDULE (GUIDED, m)** – итерации цикла делятся на блоки, размер которых уменьшается. Наименьший размер блока определяется в **m**. Как только нить закончит выполнение очередного блока, она динамически получает следующий блок итераций для выполнения.
- **SCHEDULE (RUNTIME) –** распределение итераций по нитям может меняться во время вычислений в зависимости от значения переменной окружения **OMP_SCHEDULE.**
- **ORDERED**

Этот атрибут используется тогда, когда необходимо внутри *цикла do* иметь ORDERED-секции. Внутри этих секций код выполняется в порядке, который был бы при последовательном исполнении. ORDERED-секции определяются с помощью директивы **ORDERED**.

По умолчанию при завершении директивы **DO** все нити дожидаются последней. Если же определен атрибут **NOWAIT**, нити не синхранизируют свою работу в конце параллельного выполнения цикла.

!$OMP PARALLEL DO [*атрибут*[[,] *атрибут*] …]
цикл do
!$OMP END PARALLEL DO

Директива **PARALLEL DO** позволяет более кратко записать *параллельную область*, которая содержит единственную **DO**-директиву. Семантика полностью идентична **PARALLEL**-директиве и следующей за ней **DO**-директиве. Атрибуты этой директивы совпадают с атрибутами **PARALLEL**-директивы и **DO**-директивы, описанными выше.

!$OMP SECTIONS [*атрибут*[[,] *атрибут*] …]
[**!$OMP SECTION**]
блок операторов программы

[**!$OMP SECTION**
блок операторов программы]
…
!$OMP END SECTIONS [NOWAIT]

Директива **SECTIONS** обрамляет параллельную секцию программы. Вложенные секции программы, задаваемые директивами SECTION, распределяются между нитями. Атрибутами этой директивы могут быть:

- **PRIVATE** (*список*)
- **REDUCTION** ({*оператор*|*встроенная функция*}:*список*)
- **FIRSTPRIVATE** (*список*)
- **LASTPRIVATE** (*список*)

По окончании параллельного цикла или блока параллельных секций нить, которая выполнила последнюю итерацию цикла или последнюю секцию блока, обновляет значение переменной из *списка*.

!$OMP SINGLE [*атрибут*[[,] *атрибут*] …]
блок операторов программы
!$OMP END SINGLE [NOWAIT]

Директива **SINGLE** определяет, какая часть блока операторов программы должна быть выполнена только одной нитью. Нити, которые не участвуют в выполнении операторов в блоке **SINGLE**, ожидают завершения этого блока в том случае, если не установлен атрибут **NOWAIT**. Атрибутами этой директивы могут быть:

- **PRIVATE** (*список*)
- **FIRSTPRIVATE** (*список*)

!$OMP MASTER
блок операторов программы
!$OMP END MASTER

Обрамляет блок операторов программы, который должен выполняться только главной нитью.

!$OMP CRITICAL [(*имя*)]
блок операторов программы
!$OMP END CRITICAL [(*имя*)]

Директива **CRITICAL** разрешает доступ к выделенному блоку только для одной нити в любой момент времени. Каждая оставшаяся нить приостанавливает выполнение программы перед этой ди-

рективой до тех пор, пока выделенный блок операторов программы не освободится.

!$OMP BARRIER

Директива **BARRIER** синхронизирует работу всех нитей. Каждая нить, выполнение которой достигло данной точки, ждет до тех пор, пока все нити примут это состояние.

!$OMP FLUSH (*список*)

Директива **FLUSH** задает точку синхронизации, в которой значения переменных, указанных в *списке* и видимых из данной нити, записываются в память. Тем самым обеспечивается согласование содержимого памяти, доступного разным нитям. Неявно **FLUSH** присутствует в следующих директивах: **BARRIER**, **CRITICAL**, **END CRITICAL**, **END DO**, **END PARALLEL**, **END SECTIONS**, **END SINGLE**, **ORDERED**, **END ORDERED**.

!$OMP ATOMIC

Объявляет операцию атомарной (при выполнении атомарной операции одновременный доступ к памяти по записи разных нитей запрещен). Применяется только к оператору, непосредственно следующему после данной директивы. Он может иметь следующий вид:

x = x {+|−||/|.AND.|.OR.|.EQV.|.NEQV.} expression*
x = expression {+|−||/|.AND.|.OR.|.EQV.|.NEQV.} x*
x = {MAX|MIN|IAND|IOR|IEOR} (x, expression)
x = {MAX|MIN|IAND|IOR|IEOR} (expression, x)

Здесь *expression* – скалярное выражение, не содержащее *x*.

!$OMP ORDERED
блок операторов программы
!$OMP END ORDERED

Обеспечивает сохранение того порядка выполнения итераций цикла, который соответствует последовательному выполнению программы.

2.4.2 Runtime-процедуры и переменные окружения

В целях создания переносимой среды запуска параллельных программ в OpenMP определен ряд переменных окружения, контроли-

рующих поведение приложения. В OpenMP предусмотрен также набор библиотечных процедур, которые позволяют:

- во время исполнения контролировать и запрашивать различные параметры, определяющие поведение приложения (такие как число нитей и процессоров, возможность вложенного параллелизма); процедуры назначения параметров имеют приоритет над соответствующими переменными окружения;
- использовать синхронизацию на базе замков (locks).

2.4.3 Переменные окружения

В UNIX переменные окружения задаются следующим образом: export **ПЕРЕМЕННАЯ** = значение.

OMP_SCHEDULE

Эта переменная определяет способ распределения итераций в цикле между нитями.

OMP_NUM_THREADS

Определяет число нитей для исполнения параллельных областей приложения.

OMP_DYNAMIC

Разрешает (.TRUE.) или запрещает (.FALSE.) динамическое распределение итераций между нитями.

OMP_NESTED

Разрешает или запрещает вложенный параллелизм.

2.4.4 Процедуры для контроля/запроса параметров среды исполнения

SUBROUTINE OMP_SET_NUM_THREADS (THREADS)
INTEGER THREADS

Позволяет назначить максимальное число нитей для использования в следующей параллельной области (если это число разрешено менять динамически). Вызывается из последовательной области программы.

INTEGER FUNCTION OMP_GET_MAX_THREADS ()

Возвращает максимальное число нитей, которое можно использовать в параллельной части программы.

INTEGER FUNCTION OMP_GET_NUM_THREADS ()

Возвращает фактическое число нитей в параллельной области программы.

INTEGER FUNCTION OMP_GET_THREAD_NUM ()

Возвращает идентификатор нити, из которой вызывается данная функция.

INTEGER FUNCTION OMP_GET_NUM_PROCS ()

Возвращает число процессоров, доступных приложению.

LOGICAL FUNCTION OMP_IN_PARALLEL()

Возвращает .TRUE., если функция вызвана из параллельной области программы.

SUBROUTINE OMP_SET_DYNAMIC(FLAG)
LOGICAL FLAG
LOGICAL FUNCTION OMP_GET_DYNAMIC()

Устанавливает/запрашивает состояние флага, разрешающего динамически изменять число нитей.

SUBROUTINE OMP_GET_NESTED(NESTED)
INTEGER NESTED
LOGICAL FUNCTION OMP_SET_NESTED()

Устанавливает/запрашивает состояние флага, разрешающего вложенный параллелизм.

2.4.5 Процедуры для синхронизации на базе замков

В качестве замков используются общие переменные типа INTEGER (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации.

SUBROUTINE OMP_INIT_LOCK(var)
SUBROUTINE OMP_DESTROY_LOCK(var)

Инициализирует замок, связанный с переменной var.

SUBROUTINE OMP_SET_LOCK(var)

Заставляет вызвавшую нить дождаться освобождения замка, а затем захватывает его.

SUBROUTINE OMP_UNSET_LOCK(var)

Освобождает замок, если он был захвачен вызвавшей нитью.

LOGICAL FUNCTION OMP_TEST_LOCK(var)

Пробует захватить указанный замок. Если это невозможно, возвращает .FALSE.

2.4.6 Примеры

Рассмотрим параллельную OpenMP-программу, выполняющую умножение матриц.

```
Program matrix
!подключение модуля
      USE IFPORT
!задание размерности матриц
      Parameter (n=1000)
      Integer(4) i, j, k, l, m, iseed
      Double precision A(n,n), B(n,n), C(n,n), tm(3)
! определение типа переменных блока распараллеливания
!$    integer omp_get_num_threads
!$    double precision omp_get_wtime
!$    double precision tm0, tm1
!  вывод на экран количества используемых нитей
!$omp parallel
!$     write(*,*) 'OpenMP-parallel with', omp_get_num_threads()
!$omp end parallel
! подготовка датчика случайных чисел для задания значений
! матриц
      iseed = 100
      CALL SEED(1995)
      do i=1,n
       do j=1,n
        A(i,j) = RAN(iseed)*100
        B(i,j) = RAN(iseed)*100
        C(i,j) = 0
       end do
      end do
! фиксируется время начала счета omp-блока
!$     tm0=omp_get_wtime()
      write(*,*) 'Calculation... '
! начало области распараллеливания
!$omp parallel
! распараллеливание цикла
!$omp do private (i,j,k)
  do j=1,n
      do k=1,n
          do i=1,n
```

```
        C(i,j)=C(i,j) + A(i,k) * B(k,j)
          end do
        end do
      end do
!$omp end do
! конец области распараллеливания

!$omp end parallel
! фиксируется время завершения счета omp-блока
!$     tm1=omp_get_wtime()
!$     write(*,*) tm1-tm0
       End
```

Использование библиотеки OpenMP Для вычисления определенного интеграла.

Данный пример идентичен первому примеру из раздела (*3.6.3 Вычисление определенного интеграла).* Различие заключается только в числе разбиений области интегрирования.

```
      Program example3aOMP
      implicit none
      integer i, n, s
      double precision sum, a, b, f, x
      parameter (a=0.d0, b=1.d0)
! подынтегральная функция
       f(x)=dlog(1/x)
!
!$    integer        omp_get_num_threads
!$    double precision omp_get_wtime, tm0
! фиксируется время начала счета
!$    tm0=omp_get_wtime()
! отключение выбора количества нитей по умолчанию
!$    call  omp_set_dynamic(.false.)
! заданние числа используемых нитей
!$    call  omp_set_num_threads(2)
! вывод на экран количества используемых нитей
!$omp parallel
!$     write(*,*) 'openmp-parallel with', omp_get_num_threads()
!$omp end parallel
```

```
! число разбиений области интегрирования
      n = 100000000
      sum = 0
! начало области распараллеливания цикла переменная i -
! приватная, переменная sum обновляется всеми нитями
!$omp  parallel do
!$omp& default(shared) private(i)
!$omp& schedule(static,10)
!$omp& reduction(+:sum)
      do i = 1,n
        x = a+(b-a)*(i-0.5d0)/n
        sum = sum + f(x)
      end do
!$omp  end parallel do
! конец области распараллеливания
! вычисление итогового результата и печать на экран
      sum = sum/n
      write (6,*) 'result= ',sum,' error= ',1-sum
! фиксируется время окончания счета и выдается время работы
! программы
!$     tm0=omp_get_wtime()-tm0
!$     write(6,*) 'time of calculation', tm0
    end
```

2.5 Результаты применения OpenMP на многоядерных системах

Вычислительный узел кластера ТГУ СКИФ Cyberia построен с использованием двух процессоров Intel Xeon 5150 с двухъядерной архитектурой, что позволяет провести испытания по запуску OpenMP-приложений одновременно для четырех параллельных потоков (нитей). В качестве тестовой программы рассматривалась программа в приведенном выше примере. Команда для компиляции OpenMP-приложения имеет следующий вид:

ifort –openmp –O2 example.f –o example.exe

Перед запуском необходимо указать число параллельных потоков выполнения OpenMP-программы, для чего требуется изменить

значение переменной системного окружения OMP_NUM_THREADS. Например, так:

export OMP_NUM_THREADS=4

После этого запуск исполняемой программы осуществлялся по команде:

qsub ./script.sh

Скрипт *script.sh* имеет вид:
#!/bin/sh
#PBS -l nodes=1:ppn=4
#PBS -v OMP_NUM_THREADS
cd $PBS_O_WORKDIR
ulimit -s unlimited
./example.exe

При перемножении матриц размерности 1000x1000 (рис. 2.1) было получено, что при распараллеливании матричных вычислений на два или четыре потока получается ускорение в два и четыре раза соответственно по сравнению с обычной последовательной программой.

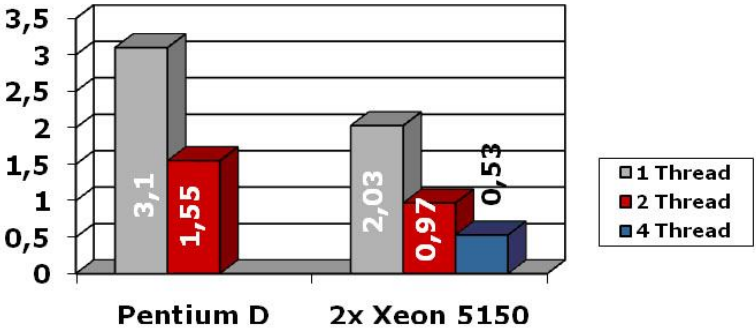

Рис. 2.1 Процессорное время при умножении матриц 1000x1000

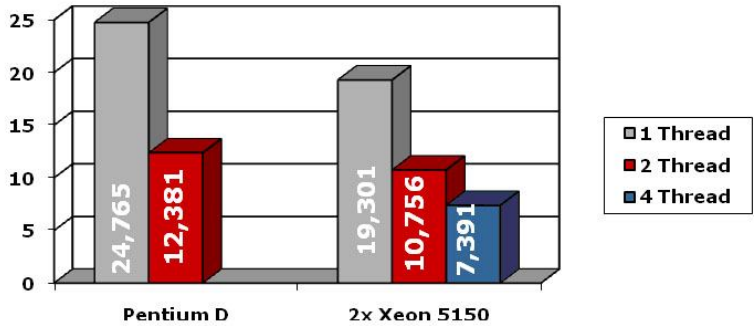

Рис. 2.2 Процессорное время при умножении матриц 2000x2000

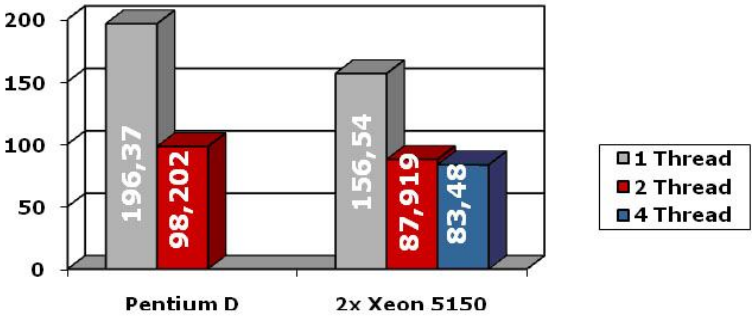

Рис. 2.3 Процессорное время при умножении матриц 4000x4000

Далее были произведены замеры времени вычислений для квадратных матриц размерности 2000x2000 и 4000x4000 (рис. 2.2 и 2.3). В рассмотренных случаях на двух потоках так же получено ускорение приблизительно в два раза, однако на четырех потоках с увеличением размерности задачи ускорение уменьшается. Так, при перемножении матриц размерности 4000x4000 время вычислений при распараллеливании на два и четыре потока практически одинаково. Возможно, данный эффект обусловлен используемой архитектурой вычислительных узлов кластера (размером кэш-памяти).

Время и ускорение программы вычисления определенного интеграла, написанной с использованием OpenMP представлены в табл. 2.2. Расчеты выполнены на кластере ТГУ Скиф-Cyberia. Достигнутое ускорение позволяет говорить о высокой эффективности использования OpenMP для подобного класса задач.

Таблица 2.2 Время и ускорение программы вычисления определенного интеграла

	1	2	4
Время, с	5,07	2,54	1,3
Ускорение	1,00	1,996	3,9

2.6 Intel Math Kernel Library

Библиотека **Intel Math Kernel Library (MKL)** представляет собой набор функций линейной алгебры, быстрого преобразования Фурье и векторной математики для разработки научного и инженерного ПО. Она представлена в вариантах для **Windows** и **Linux**, есть версия для **Linux-**кластеров. Библиотеки **MKL** состоят из нескольких двоичных файлов, каждый из которых оптимизирован для определенного семейства процессоров **Intel**, включая **Intel Itanium 2**, **Intel Xeon**, **Intel Pentium III** и **Intel Pentium 4**.

На стадии выполнения **MKL** автоматически определяет модель процессора и запускает соответствующую версию вызываемой функции, что гарантирует максимальное использование возможностей процессора и максимально возможную производительность.

Все библиотеки **MKL** поддерживают работу в многопоточном режиме, а ключевые функции **LAPACK**, **BLAS** (третьего уровня) и дискретных преобразований Фурье (DFT) допускают распараллеливание по стандарту **OpenMP**.

Так как библиотека **MKL** максимально оптимизирована под процессоры **Intel** и учитывает их архитектуру, использование компонентов библиотеки более предпочтительно, чем ручное написание кода. Для демонстрации вышесказанного сравним пользовательскую программу блочного умножения матриц с процедурой **DGEMM** из пакета **BLAS3**. Данная библиотека позволяет производить вычисления следующего вида:

$$C = \alpha \cdot A \cdot B + \beta \cdot C,$$

где A, B, C – соответствующие по размерности матрицы с базовым типом двойной точности, а α и β – некоторые коэффициенты.

Зададим значения коэффициентам α и β соответственно 1 и 0 и получим простое перемножение матрицы A на B. Результат времени умножения матриц с использованием **DGEMM** будем сопоставлять с временем, полученным при оптимизированном перемножении блочных матриц на основе пользовательского кода.

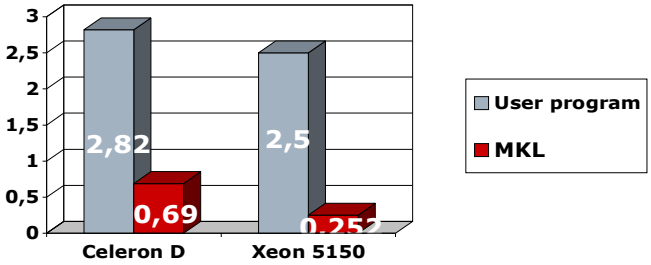

Рис. 2.4 Процессорное время при умножении матриц 1000x1000

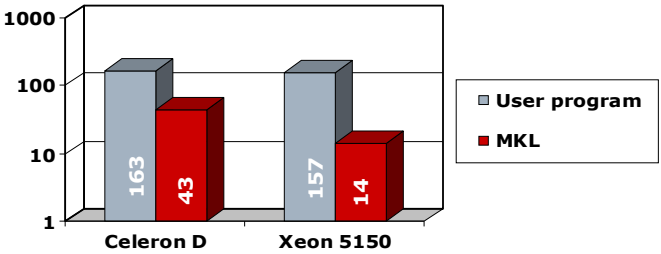

Рис. 2.5 Процессорное время при умножении матриц 4000x4000

На рис. 2.4–2.5 показано сравнение процессорного времени перемножения матриц, полученного с использованием обычного алгоритма блочных скалярных произведений, и специализированных процедур **MKL**. На рис. 2.4 приведены результаты при перемножении матриц размерностью 1000x1000 элементов. Видно, что применение **MKL** позволяет получить результат в 4-5 раз быстрее. Подобные сравнительные расчеты проводились и с матрицами 4000x4000 элементов (рис. 2.5). Из рисунков видно, что обнаруженный эффект повышения быстродействия вычислений

сохраняется. Таким образом, использование специализированных библиотек существенно ускоряет процесс вычислений.

Выводы

Компилятор Intel Compiler 9.1 имеет большое количество опций, которые позволяют максимально оптимизировать задачу под архитектуру компьютера.

Применение технологии OpenMP для матричных вычислений на двух потоках дало ускорение почти в два раза, а на четырех потоках – в четыре раза, однако в связи с архитектурными особенностями процессоров Xeon 5150 четырехкратное ускорение можно получить лишь при перемножении матриц относительно небольших размерностей (1000x1000).

Нужно стараться заменять разработанный пользователем код процедурами из специализированных библиотек. Благодаря этому можно получить ускорение в 4-5 раз за счет максимального использования архитектуры процессора.

Все вычисления производились с помощью компилятора Intel Compiler 9.1 на процессорах Intel различной производительности.

3 ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ СТАНДАРТА MPI

Наиболее общим подходом при создании параллельных программ в настоящее время является применение библиотеки передачи сообщений, где процессы используют обращения к библиотеке **MPI** (**Message Passing Interface**), чтобы обмениваться сообщениями (информацией) с другими процессами. Механизм передачи сообщений позволяет процессам, запущенным на многопроцессорной вычислительной системе, совместно решать большие задачи.

MPI – это стандарт передачи сообщений, который разрабатывался группой 60 человек из 40 организаций США и Европы. Первый вариант стандарта **MPI 1.0** был опубликован в 1994 г. Большинство современных реализаций соответствуют стандарту **MPI 1.1** (опубликован в 1995 г.). В 1997 г. был опубликован стандарт **MPI 2.0**, однако он не получил широкого распространения. MPI пригоден для различных платформ, начиная с массивно-параллельных систем (например, IBM SP2, Cray T3D, Intel Paragon) и заканчивая сетями рабочих станций (Sun4, Dec Alpha). Основной целью, преследуемой при создании MPI, было разработать практический, переносимый, эффективный и удобный стандарт для передачи сообщений между процессами.

Стандарт MPI включает 129 функций для:

- инициализации и закрытия параллельной части приложения;
- приема и передачи сообщений между отдельными процессами;
- осуществления коллективного взаимодействия процессов;
- работы с группами процессов и коммуникаторами;
- определения производных (составных) типов данных;
- создания виртуальных топологий для обеспечения более простого взаимодействия процессов.

Однако начинающему пользователю при написании своих параллельных программ можно обойтись их малым подмножеством (например, минимально — шестью).

3.1 Основные понятия MPI

К базовым понятиям MPI относятся *процесс, группа процессов* и *коммуникатор*. Процесс – это исполнение программы одним процессорным элементом, на котором загружен MPI. Процессы объе-

диняются в группы с единой областью связи, внутри которой каждый процесс имеет свой уникальный номер в диапазоне от 0 до N–1, где N – количество процессов в группе. Для взаимодействия процессов в группе используется коммуникатор, который реализует передачу сообщений (обмен данными) между процессами и их синхронизацию. Коммуникатор осуществляет обмены только внутри области связи группы, с которой он ассоциируется.

Обычно MPI-программа для многопроцессорных вычислительных систем пишется на языке C, C++ или FORTRAN с использованием коммуникационных функций библиотеки MPI. Запуск на выполнение MPI-программы представляет собой одновременный запуск совокупности параллельных процессов, количество которых определяет пользователь при запуске. Параллельная часть программы начинается с инициализации MPI. При этом все активированные процессы объединяются в группу с коммуникатором, который имеет имя MPI_COMM_WORLD. Далее средствами MPI в программу передается число активированных процессов, каждому из них присваивается свой номер. Процессы, как правило, отличаются тем, что каждый отвечает за исполнение своей ветви параллельной MPI-программы.

3.2 Начала MPI и программа `Hello World'

Полный стандарт MPI состоит из 129 функций. Однако начинающему пользователю MPI достаточно научиться применять в практической работе лишь несколько из них (6 или 24 – в зависимости от квалификации). Без чего действительно нельзя обойтись, так это без функций, передающих и принимающих сообщения. Выполнение следующих правил позволит легко сконструировать большинство MPI-программ.

- Все MPI-программы должны включать заголовочный файл (в C-программе *mpi.h*; в FORTRAN-программе *mpif.h*).
- Все MPI-программы должны вызывать подпрограмму MPI_INIT как первую вызываемую подпрограмму MPI для того, чтобы инициализировать MPI.
- Большинство MPI-программ вызывает подпрограмму MPI_COMM_SIZE, чтобы определить размер активированной виртуальной машины, т.е. количество запущенных процессов size.
- Большинство MPI-программ вызывает MPI_COMM_RANK,

чтобы определить номер каждого активированного процесса, который находится в диапазоне от 0 до size-1.

- В каждой MPI-программе активируются процессы, характер выполнения которых может меняться по условию. Взаимодействие процессов производится путем передачи сообщений между отдельными процессами с помощью вызова функций MPI_SEND и MPI_RECV.
- Все MPI-программы должны обращаться к MPI_FINALIZE как к последнему вызову функций из MPI-библиотеки.

Таким образом, можно написать достаточное количество MPI-программ, пользуясь только следующими шестью функциями: MPI_INIT, MPI_COMM_SIZE, MPI_COMM_RANK, MPI_SEND, MPI_RECV, MPI_FINALIZE.

Однако для более удобного программирования следует изучить и несколько коллективных коммуникационных функций. Эти функции позволяют взаимодействовать нескольким процессам одновременно.

Ниже представлены Fortran и C-версии параллельной MPI-программы ‘Hello World’. Ее можно взять в качестве начального примера при освоении правил написания MPI-программ. Эти простые программы являются SPMD (Single Program Multiple Data) программами, где все процессы представляют собой запуск одного и того же кода, но могут выполнять различные действия в зависимости от значений данных (например, номера процесса). В представленной ниже программе главный процесс (rank = 0) посылает сообщение из символов ‘Hello World’ остальным процессам (rank > 0). Они принимают это сообщение и печатают его на экране.

Если эта программа будет запущена командой

mpirun – np 3 ./hello ,

то на экране может быть напечатано следующее:

process 2 : Hello, world!
process 0 : Hello, world!
process 1 : Hello, world!

C-версия программы ‘Hello World’:

```
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
 int rank, size, tag, rc, i;
 MPI_Status status;
 char message[20];
 rc = MPI_Init(&argc, &argv);
 rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
 rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 tag = 8;
 if (rank == 0) {
      strcpy(message, "Hello, world!");
  for (i=1; i<size; i++)
   rc = MPI_Send(message, 13, MPI_CHAR, i, tag,
                 MPI_COMM_WORLD);
 }
  else
   rc = MPI_Recv(message, 13, MPI_CHAR, 0, tag,
                 MPI_COMM_WORLD, &status);
   printf( "process %d : %s\n", rank, message);
   rc = MPI_Finalize();
}
```

Fortran-версия программы ‘Hello World’:

```
 program hello
 include 'mpif.h'
 integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
 character(13) message
 call MPI_INIT(ierror)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
 tag = 8
 if (rank .eq. 0) then
   message = 'Hello, world!'
   do i=1, size-1
    call MPI_SEND(message, 13, MPI_CHARACTER, i, tag,
```

```
$                       MPI_COMM_WORLD, ierror)
  end do
 else
  call MPI_RECV(message, 13, MPI_CHARACTER, 0,
$                     tag, MPI_COMM_WORLD, status, ierror)
 end if
 print*, 'process', rank, ':', message
 call MPI_FINALIZE(ierror)
 end
```

3.3 Синтаксис базовых функций MPI

Инициализация MPI осуществляется при вызове функции MPI_INIT:

```
C:      int MPI_Init(int* argc, char*** argv)
FORTRAN:  MPI_INIT(ierror)
          INTEGER ierror
```

Возвращается код ошибки ierror. Если ierror=0, то оператор проработал успешно.

Завершение MPI производится при вызове MPI-функции MPI_FINALIZE:

```
C:      int MPI_Finalize(void)
FORTRAN:  MPI_FINALIZE(ierror)
          INTEGER ierror
```

Определение числа процессов в группе COMM:

```
C:      int MPI_Comm_size(MPI_Comm comm,int* size)
FORTRAN:  MPI_COMM_SIZE(COMM, size, ierror)
          INTEGER COMM, size, ierror
```

Функция возвращает количество процессов size в области связи с коммуникатором COMM.

Определение номера процесса в группе COMM:

C: int MPI_Comm_rank(MPI_Comm comm,int*rank)
FORTRAN: MPI_COMM_RANK(COMM, rank, ierror)
INTEGER COMM, rank, ierror

Функция возвращает номер процесса, вызвавшего ее, из области связи с коммуникатором COMM.

Посылка сообщения процессу:

C: int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm)

FORTRAN: MPI_SEND(buf, count, datatype, dest, msgtag, COMM, ierror)
<type> buf(*)
INTEGER count, datatype, dest, msgtag, COMM, ierror

Функция осуществляет блокирующую посылку сообщения buf с идентификатором msgtag, состоящего из count элементов типа datatype, процессу с номером dest в области связи с коммуникатором COMM. Все параметры этой функции являются входными за исключением ierror, который возвращает код ошибки. Тип передаваемых элементов datatype должен указываться с помощью предопределенных констант (см. табл. 3.1, 3.2).

Таблица 3.1 Соответствие между типами C и типами MPI

Тип C	Тип MPI
signed char	MPI_CHAR
signed short int	MPI_SHORT
signed int	MPI_INT
signed long int	MPI_LONG
unsigned char	MPI_UNSIGNED_CHAR
unsigned short int	MPI_UNSIGNED_SHORT
unsigned long int	MPI_UNSIGNED_LONG
Float	MPI_FLOAT
Double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE
	MPI_BYTE
	MPI_PACKED

Таблица 3.2 Соответствие между типами FORTRAN 77 и типами MPI

Тип FORTRAN77	Тип MPI
INTEGER	MPI_INTEGER
REAL	MPI_REAL
DOUBLE PRECISION	MPI_DOUBLE_PRECISION
COMPLEX	MPI_COMPLEX
LOGICAL	MPI_LOGICAL
CHARACTER(1)	MPI_CHARACTER
	MPI_BYTE
	MPI_PACKED

Прием сообщения от процесса:

C: int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Status *status)

FORTRAN MPI_Recv(buf, count, datatype, source, msgtag, COMM,STATUS, ierror)
<type> buf(*)
INTEGER count, datatype, dest, msgtag, COMM, ierror,STATUS(MPI_STATUS_SIZE)

Функция осуществляет прием сообщения buf с идентификатором msgtag от процесса source с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения count. Тип получаемых данных datatype должен указываться с помощью именованных констант MPI. STATUS – целочисленный массив, содержащий параметры принимаемого сообщения.

Библиотека MPI также содержит и неблокирующие функции передачи сообщений между двумя процессами: MPI_Isend, MPI_Irecv. В отличие от блокирующих, которые приостанавливают вызвавший их процесс до тех пор, пока операция передачи не будет завершена, неблокирующие подразумевают совмещение операций обмена с другими операциями.

Обычно для оценки эффективности и ускорения работы параллельной программы необходимо определить время, затрачиваемое на ее выполнение. Для этой цели в библиотеке MPI предусмотрена функция MPI_Wtime().

C: double MPI_Wtime(void)
FORTRAN: DOUBLE PRECISION MPI_Wtime()

Функция возвращает астрономическое время в секундах, прошедшее с некоторого фиксированного момента в прошлом.

В библиотеке MPI вводятся собственные типы. Это было продиктовано тем, что языки программирования на разных машинных платформах для одних и тех же типов имеют различное представление в байтах. А поскольку MPI ориентирована на переносимость программных приложений, введение собственных типов дает возможность запуска процессов на различных платформах с автоматическим преобразованием данных при пересылках. В библиотеке MPI для всех стандартных типов данных определены соответствующие константы, позволяющие идентифицировать эти типы в функциях передачи и приема данных.

3.4 Некоторые функции коллективного взаимодействия процессов

Некоторые функции библиотеки MPI позволяют программисту дать команду подгруппе процессов виртуальной машины. Члены подгруппы, принадлежащие одному коммуникатору, идентифицируются своими номерами, и отдельный процесс может быть членом более чем одной подгруппы. Коллективные функции облегчают решение таких задач, как синхронизация процессов, глобальные операции редукции по подгруппе процессов, расщепление данных по процессам и сборка данных на одном процессе. Здесь рассматриваются функции MPI_BARRIER, MPI_BCAST и MPI_REDUCE. Другие коллективные функции – MPI_SCATTER, MPI_GATHER, MPI_ALLREDUCE – будут описаны ниже, и их использование показано на примерах в следующей части.

Функция MPI_BARRIER блокирует вызвавший ее процесс до тех пор, пока все остальные члены группы с коммуникатором COMM не вызовут ее.

C: int MPI_Barrier(MPI_Comm comm)
FORTRAN: MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR

MPI_BCAST рассылает сообщение buf процесса с номером root остальным процессам группы COMM, в том числе и себе. Каждый процесс получает копию count элементов типа datatype.

C: int MPI_Bcast(void* buf, int count, MPI_Datatype datatype,int root, MPI_Comm comm)
FORTRAN: MPI_BCAST(buf, count, datatype, root, COMM, ierror)
<type> buf(*)
INTEGER count, datatype, root, COMM, ierror

MPI_REDUCE комбинирует элементы sendbuf каждого процесса в recvbuf процесса root. Комбинация подразумевает следующие операции: нахождение максимального (признак операции MPI_MAX), определение минимального (признак операции MPI_MIN), нахождение суммы (MPI_SUM), вычисление произведения (MPI_PROD).

C: int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
FORTRAN: MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, COMM,ierror)
<type> sendbuf(*), recvbuf(*)
INTEGER count, datatype, op, root, COMM, ierror

Здесь все параметры являются входными за исключением recvbuf и ierror. В параметре recvbuf процесса root собирается результат операции op типа datatype, полученный в результате комбинации count элементов sendbuf каждого процесса группы с коммуникатором COMM.

3.5 Другие возможности MPI

Если необходимо другим процессам группы передать информацию, расположенную в несмежных областях памяти, или сообщение, содержащее данные различных типов, то можно использовать упаковку сообщений или переопределять производные типы данных.

На основе функций библиотеки MPI допустимо создавать виртуальные топологии для того, чтобы облегчить размещение элементов задачи по процессорам и эффективно выполнять общие операции, связывающие соседние процессоры, например в двумерной декартовой решетке.

3.6 Примеры параллельных MPI-программ на языке FORTRAN

3.6.1 Идентификация процессов

Рассмотрим текст следующей программы:

```
Program Example1

  Implicit None
  Include 'mpif.h'

  Integer Rank, Size, Ierr

C  Инициализация MPI и определение процессорной конфигурации
  Call MPI_INIT(Ierr)
  Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
  Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

  Write(6,*) 'Process ',Rank,' of ',Size,' is ready to work.'

C  Завершение работы функций MPI
  Call MPI_FINALIZE(Ierr)

  End
```

Данный пример является простейшей программой, использующей базовые функции MPI. Эту программу нельзя в полном смысле назвать параллельной, так как в ней не реализованы взаимодействия между процессами, однако каждый процесс получает информацию как об общем числе процессов, запустивших программу, так и о своем номере в группе процессов с коммуникатором MPI_COMM_WORLD.

В приведенном тексте программы вначале каждый процесс производит инициализацию MPI вызовом процедуры MPI_INIT, затем определяет общее число процессоров и свой номер, вызывая MPI_COMM_SIZE и MPI_COMM_RANK, которые возвращают значения общего числа и номера в переменные Size и Rank соответственно. Параметр MPI_COMM_WORLD является предопределенным коммуникатором, объединяющим в единую область связи все процессы приложения. После инициализации MPI процессы в группе упорядочиваются и нумеруются от 0 до Size-1, где Size – общее число процессов.

После выполнения основной части программы (в данном случае одного оператора вывода Write) необходимо закрыть все (каждый в отдельности) MPI-процессы, что и производится вызовом MPI_FINALIZE.

В результате работы этой программы каждый процесс выведет строку о своей готовности к работе. После запуска программы, например на 10 процессорах, пользователь, возможно, увидит следующее:

Process 0 of 10 is ready to work.
Process 9 of 10 is ready to work.
Process 4 of 10 is ready to work.
Process 1 of 10 is ready to work.
Process 8 of 10 is ready to work.
Process 3 of 10 is ready to work.
Process 6 of 10 is ready to work.
Process 5 of 10 is ready to work.
Process 7 of 10 is ready to work.
Process 2 of 10 is ready to work.

Можно обратить внимание на то, в какой последовательности на экране выводятся строки. Такой порядок может показаться хаотичным, но это не совсем так. Первым свою информацию всегда выведет процессор с номером «0». И только после этого он предоставит возможность вывода остальным процессам, порядок которых может варьироваться. Это связано с особенностью эмуляции экранного вывода многопроцессорных систем (процессы не имеют в своем распоряжении экрана), и при проектировании сложных программ следует учитывать то, что ваша MPI-система сконфигурирована таким образом, что процессы, отличные от «0»-процесса, смогут

отобразить информацию на терминальном экране лишь при полном окончании выполнения программы на «0»-процессе. Если же что-нибудь необходимо отобразить еще в ходе выполнения программы, то следует переслать эту информацию «0»-процессу и поручить ему вывести ее на экран (использование такого принципа можно найти в следующем примере).

3.6.2 Коммуникационные операции между двумя процессами

```
      Program Example2

       Implicit None
       Include 'mpif.h'
       Integer Rank, Size, Ierr, I, IRank, Status(MPI_STATUS_SIZE)

       Call MPI_INIT(Ierr)
       Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
       Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

       If (Rank.eq.0) Then

C   Эти операторы выполняет «0»-процесс (он принимает)...
         Do I = 1,Size-1
           Call MPI_RECV(IRank, 1, MPI_INTEGER, I, 0,
     $                   MPI_COMM_WORLD,Status, Ierr)
          Write(6,*) 'Process',IRank,' of',Size,' is ready to work.'
         End Do
         Write(6,*) 'Process',Rank,' have finished receiving now.'
       Else

C   ... а все остальные процессы отправляют
         Call MPI_SEND(Rank, 1, MPI_INTEGER, 0, 0,
     $                       MPI_COMM_WORLD,Ierr)
       End If
       Call MPI_FINALIZE(Ierr)
       Stop
      End
```

Данный пример реализует простейшие коммуникационные операции типа «точка-точка» – блокирующие функции передачи и

приема сообщений MPI_SEND и MPI_RECV. В основной части программы расположен условный оператор, который разделяет операторы, выполняемые «0»-процессом, от выполняемых другими процессами. Процессы получают различные задания: все, кроме «0»-процесса, производят посылку целого значения своего номера Rank в адресное пространство «0»-процесса, в то время как «0»-процесс в цикле получает значения, отправленные другими процессами, выводит эти значения на экран и затем докладывает об окончании работы.

Запуск программы на 10 процессорах приведет к выводу следующей последовательности строк:

Process 1 of 10 is ready to work.
Process 2 of 10 is ready to work.
Process 3 of 10 is ready to work.
Process 4 of 10 is ready to work.
Process 5 of 10 is ready to work.
Process 6 of 10 is ready to work.
Process 7 of 10 is ready to work.
Process 8 of 10 is ready to work.
Process 9 of 10 is ready to work.
Process 0 have finished receiving now.

Этот пример отличается от предыдущего и тем, что в нем процессы заканчивают свою работу не в одно время, а следующим образом: первым свою работу закончит процесс с номером «1» – он дождется приема «0»-процессом своего сообщения, и на этом его работа закончится. Потом закончит работу «2»-процесс, затем «3»-процесс и т.д. Самым последним завершит работу, конечно, «0»-процесс. Если необходимо изменить порядок и заставить все процессы ждать, пока «0»-процесс не справится со своей работой, то следует поставить вызов функции синхронизации

Call MPI_BARRIER(MPI_COMM_WORLD, Ierr)

непосредственно перед завершением работы функций MPI – это приведет к тому, что ни один процесс не продолжит (а в данном случае не завершит) выполнение программы до тех пор, пока все другие процессы не вызовут эту функцию.

Следует заметить, данная программа обладает не самым лучшим свойством – сначала процессы ждут, когда сообщения обработают, а затем, возможно, ожидают завершения работы «0»-процесса. Такую программу можно назвать плохо сбалансированной и даже неэффективной. Далее будет показано, как можно избежать различного рода простоев и тем самым ускорить выполнение параллельных программ.

3.6.3 Вычисление определенного интеграла

Следующая серия примеров посвящена различным способам параллельной реализации численного определения значения интеграла по квадратурной формуле средних прямоугольников с заданным разбиением области интегрирования.

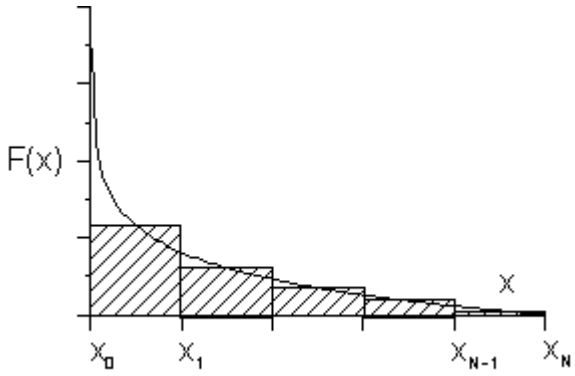

Рис. 3.1 Приближенное вычисление интеграла
по формуле средних прямоугольников

Пусть ставится задача вычисления значения интеграла вида $\int_a^b F(x)dx$. Рассмотрим $F(x) = \ln(1/x)$, а в качестве пределов интегрирования $a = 0$, $b = 1$. Поскольку $\int \ln(1/x)dx = x\left(\ln(1/x)+1\right)+C$, очевидно, что интеграл при данных пределах интегрирования является несобственным и имеет значение, равное 1. Так как рассматривается несобственный интеграл ($\lim_{x\to 0} F(x) = \infty$), численное определение его

значения с наперед заданной точностью становится непростой вычислительной задачей.

Построим в области интегрирования равномерную сетку $\varpi_{\mathrm{N}} = \left\{ x_k \mid x_k = \frac{k}{N}, k = \overline{0, N} \right\}$ и воспользуемся формулой средних прямоугольников (рис. 3.1). При данном подходе приближенное значение искомого интеграла определяется по формуле $\int_0^1 F(x)dx \approx \frac{1}{N}\sum_{k=0}^{N-1} F\left(\frac{x_{k+1}+x_k}{2}\right)$, что позволяет не вычислять значения подынтегральной функции в узлах сетки.

Приведенные ниже примеры построены по следующему принципу: область интегрирования разбивалась на подобласти по числу используемых процессов, а исходный интеграл представлялся в виде суммы интегралов по таким частичным отрезкам. Схема вычислений состоит из трех этапов: во-первых, каждый процесс определяет свои границы интегрирования и число интервалов разбиения, приходящихся на данную подобласть. Во-вторых, каждый процесс вычисляет интеграл по своему частичному отрезку. На последнем, третьем этапе, происходит суммирование всех вычисленных значений «0»-процессом с получением значения интеграла. Кроме того, на «0»-процесс возлагается обязанность оценки времени, затраченного на выполнение программы, определения достигнутой точности (разность точного и приближенного значения интеграла) и вывода результата на экран.

```
      Program Example3a
        Include 'mpif.h'
        Integer Size, Rank, Ierr, I, N, Status(MPI_STATUS_SIZE)
        Double Precision Sum, GSum, A, B, time1, time2,
     $                   Al, Bl, X, F, ISum

C     Пределы интегрирования
        Parameter (A=0.d0, B=1.d0)

C     Подынтегральная функция
        F(x)=DLog(1/x)
        Call MPI_INIT(Ierr)
        Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
```

```
      Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

C   «0»-процесс засекает время
      If (Rank.eq.0) time1 = MPI_WTime()

C   Каждый процесс определяет свои пределы интегрирования
C   и число интервалов разбиения
      Al = A+(B-A)*Rank/Size
      Bl = Al+(B-A)/Size
      N = 1000000

C   Каждый процесс определяет свою частичную сумму ...
      Sum = 0
      Do I = 1,N
        X = Al+(Bl-Al)*(I-0.5d0)/N
        Sum = Sum + F(X)
      End Do

C   «0»-процесс получает частичные суммы и определяет
C   результат
      If (Rank.eq.0) Then
        GSum = Sum
        Do I=1,Size-1
          Call MPI_RECV(ISum, 1, MPI_DOUBLE_PRECISION,
     $                        i, 0, MPI_COMM_WORLD, Status, Ierr)
        GSum = GSum + ISum
      End Do

      time2 = MPI_WTime()
      GSum = GSum/(N*Size)
      Write (6,*) 'Result= ',GSum,' Error= ',1-GSum,
     $                                 ' Time=',time2 - time1
    else
      Call MPI_SEND(Sum, 1, MPI_DOUBLE_PRECISION, 0,
     $                      0, MPI_COMM_WORLD, Ierr)
    End If
    Call MPI_FINALIZE(Ierr)
   Stop
   End
```

В результате выполнения программы на экране появится строка:

Result= 0.999999913356637 Error= 8.664336337282919E-008 Time= 5.351901054382324E-002

Этот пример использует блокирующие функции пересылки MPI_SEND и MPI_RECV. Но даже используя такие простые коммуникационные функции, можно найти возможность несколько ускорить алгоритм. Сравните со следующим примером:

```
      Program Example3b

       Implicit None
       Include 'mpif.h'

       Integer Size, Rank, Ierr, I, N, Status(MPI_STATUS_SIZE)

       Double Precision Sum, GSum, A, B, time1, time2,
     $                  Al, Bl, X, F, ISum

C  Пределы интегрирования
       Parameter (A=0.d0, B=1.d0)

C  Подынтегральная функция
       F(x)=DLog(1/x)

       Call MPI_INIT(Ierr)
       Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
       Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

C  «0»-процесс засекает время
       If (Rank.eq.0) time1 = MPI_WTime()

C  Каждый процесс определяет свои пределы интегрирования
C  и число интервалов разбиения
       Al = A+(B-A)*Rank/Size
       Bl = Al+(B-A)/Size
       N = 1000000

C  Каждый процесс определяет свою частичную сумму ...
```

```
      Sum = 0
      Do I = 1,N
        X = Al+(Bl-Al)*(I-0.5d0)/N
        Sum = Sum + F(X)
      End Do
C   «0»-процесс получает частичные суммы и определяет
C    результат
      If (Rank.eq.0) Then
        GSum = Sum
        Do I=1,Size-1
          Call MPI_RECV(ISum, 1, MPI_DOUBLE_PRECISION,
     $                  MPI_ANY_SOURCE, 0,
     $                  MPI_COMM_WORLD, Status, Ierr)
          GSum = GSum + ISum
        End Do

        time2 = MPI_WTime()
        GSum = GSum/(N*Size)
        Write (6,*) 'Result=',GSum,' Error=',1-GSum,
     $                          ' Time=',time2 - time1
      else
        Call MPI_SEND(Sum, 1, MPI_DOUBLE_PRECISION, 0,
     $                  0, MPI_COMM_WORLD, Ierr)
      End If
      Call MPI_FINALIZE(Ierr)
    Stop
    End
```

Изменение коснулось лишь одного оператора – при приеме «0»-процессом сообщений от других процессов на месте параметра, отвечающего за номер процесса-отправителя, вместо явно перечисляемого индекса I появилась константа MPI_ANY_SOURCE, использование которой позволяет строить так называемые расширенные запросы. В таком варианте «0»-процесс, определенно зная общее число посылок, которые ему предстоит принять, не ожидает прихода сообщения от конкретного процесса, а принимает то сообщение, которое пришло раньше (в конце концов, не имеет же значения, в какой последовательности суммировать частичные суммы). В ряде случаев, особенно в сложных программах с различной нагруз-

кой на процессы и большими объемами пересылаемых данных, этот прием может оказаться весьма полезным.

Ниже представлена программа вычисления определенного интеграла с использованием функций неблокирующих коммуникационных операций MPI_ISEND и MPI_IRECV, а также функции ожидания завершения неблокирующих операций MPI_WAIT_ALL.

```
      Program Example3c

       Implicit None
       Include 'mpif.h'

       Integer Size, Rank, Ierr, I, N, Status(MPI_STATUS_SIZE),
     $           LStatus(0:20), MyRequest, Request(0:20)
       Double Precision Sum, GSum, A, B, time1, time2, Al,
     $                  Bl, X, F, ASum(0:20)

C    Пределы интегрирования
       Parameter (A=0.d0, B=1.d0)

C    Подынтегральная функция
       F(x)=DLog(1/x)
       Call MPI_INIT(Ierr)
       Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
       Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

C    «0»-процесс засекает время
       If (Rank.eq.0) time1 = MPI_WTime()

C    Каждый процесс определяет свои пределы интегрирования
C    и число интервалов разбиения
       Al = A+(B-A)*Rank/Size
       Bl = Al+(B-A)/Size
       N = 1000000

C    Каждый процесс определяет свою частичную сумму ...
       Sum = 0
       Do I = 1,N
         X = Al+(Bl-Al)*(I-0.5d0)/N
         Sum = Sum + F(X)
       End Do
```

```
C   ... посылает ее «0»-процессу ...
      Call MPI_ISEND(Sum, 1, MPI_DOUBLE_PRECISION, 0,
     $                    0, MPI_COMM_WORLD, MyRequest, Ierr)
C   «0»-процесс получает частичные суммы и заносит их в
C   массив ASum
      If (rank.eq.0) then
        GSum = 0
        Do I=0,Size-1
          Call MPI_IRECV(ASum(I), 1, MPI_DOUBLE_PRECISION,
     $                   I, 0, MPI_COMM_WORLD, Request(I), Ierr)
        End Do
        Call MPI_ISEND(Sum, 1, MPI_DOUBLE_PRECISION, 0,
     $                    0, MPI_COMM_WORLD, MyRequest, Ierr)
C   «0»-процесс ожидает подтверждения приема всех посылок ...
        Call MPI_Waitall(Size, Request(0), LStatus, Ierr)

C   ... и только после этого производит суммирование
        Do I=0,Size-1
          GSum = GSum + ASum(I)
        End Do

        time2 = MPI_WTime()
        GSum = GSum/(N*Size)
        Write (6,*) 'Result=',GSum,' Error=',1-GSum,
     $                                 ' Time=',time2 - time1
      End If

      Call MPI_FINALIZE(Ierr)
    Stop
    End
```

В этой программе оператор вызова Call MPI_WAIT, который выполняют все процессы, можно опустить, так как после отправки посылок процессам не о чем больше беспокоиться – рано или поздно MPI доставит сообщения (если они, конечно, будут запрошены получателем).

В том случае, если операции пересылки затрагивают все процессы, часто удобным становится использовать не коммуникационные операции типа «точка-точка», а коллективные операции. Особенно-

стью данного типа операций является то, что они имеют достаточно простой и понятный синтаксис.

В приведенной ниже программе вычисления определенного интеграла используется глобальная операция редукции с сохранением результата в адресном пространстве «0»-процесса. Значения переменных Sum пересылаются на «0»-процесс и складываются с сохранением значения результата в переменной GSum. Весь этот процесс выполняется вызовом всего одного оператора Call MPI_REDUCE.

```
      Program Example3d

        Implicit None
        Include 'mpif.h'
        Integer Size, Rank, Ierr, I, N
        Double Precision Sum, GSum, A, B, time1, time2, Al, Bl, X, F
C     Пределы интегрирования
        Parameter (A=0.d0, B=1.d0)

C     Подынтегральная функция
        F(x)=DLog(1/x)

        Call MPI_INIT(Ierr)
        Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
        Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

C     «0»-процесс засекает время
        If (Rank.eq.0) time1 = MPI_WTime()

C     Каждый процесс определяет свои пределы интегрирования
C     и число интервалов разбиения
        Al = A+(B-A)*Rank/Size
        Bl = Al+(B-A)/Size
        N = 1000000

C     Каждый процесс определяет свою частичную сумму
        Sum = 0
        Do I = 1,N
          X = Al+(Bl-Al)*(I-0.5d0)/N
          Sum = Sum + F(X)
        End Do
```

```
C «0»-процесс получает результат суммирования частичных
C  сумм
      Call MPI_REDUCE(Sum, GSum, 1,
     $              MPI_DOUBLE_PRECISION, MPI_SUM,
     $                    0, MPI_COMM_WORLD, Ierr)

      If (Rank.eq.0) Then
        time2 = MPI_WTime()
       GSum = GSum/(N*Size)
       Write (6,*) 'Result=',GSum,' Error=',1-GSum,
     $                                  ' Time=',time2 - time1
      End If

      Call MPI_FINALIZE(Ierr)

     Stop
     End
```

Если имеется необходимость получить результат глобальной редукции всеми процессами (например, в том случае, если это значение предполагается применить в дальнейших расчетах), следует использовать функцию MPI_ALLREDUCE:

```
      Call MPI_ALLREDUCE(Sum, GSum, 1,
     $              MPI_DOUBLE_PRECISION, MPI_SUM,
     $                    MPI_COMM_WORLD, Ierr)
```

Как можно заметить, при обращении к функции коллективной редукции опущен идентификатор номера процесса-получателя.

В программе вычисления определенного интеграла **Example3e** используется функция MPI_GATHER, которая производит сборку значений переменной Sum, посылаемых всеми процессами в массив Buffer «0»-процесса. После получения данных «0»-процесс производит суммирование компонент массива и рассылает результат всем остальным процессам при помощи функции рассылки MPI_BCAST.

```
     Program Example3e
      Implicit None
```

```
      Include 'mpif.h'
      Integer Size, Rank, Ierr, I, N
      Double Precision Sum, GSum, A, B, time1, time2,
     $                 Al, Bl, X, F, Buffer(0:100)
C   Пределы интегрирования
       Parameter (A=0.d0, B=1.d0)
C   Подынтегральная функция
       F(x)=DLog(1/x)
       Call MPI_Init(Ierr)
       Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
       Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

C   «0»-процесс засекает время
       If (Rank.eq.0) time1 = MPI_WTime()

C   Каждый процесс определяет свои пределы интегрирования
C   и число интервалов разбиения
      Al = A+(B-A)*Rank/Size
      Bl = Al+(B-A)/Size
      N = 1000000

C   Каждый процесс определяет свою частичную сумму ...
      Sum = 0
      Do I = 1,N
        X = Al+(Bl-Al)*(I-0.5d0)/N
        Sum = Sum + F(X)
      End Do

C   «0»-процесс получает частичные суммы со всех процессов ...
       Call MPI_GATHER(Sum, 1, MPI_DOUBLE_PRECISION,
     $                 Buffer, 1, MPI_DOUBLE_PRECISION, 0,
     $           MPI_COMM_WORLD, Ierr)
C   ... и суммирует их
      If (Rank.eq.0) Then
        GSum = 0
        Do I = 0,Size-1
          GSum = GSum+Buffer(I)
        End Do
        GSum = GSum/(N*Size)
```

```
      End If

C   «0»-процесс рассылает всем результат суммирования
      Call MPI_BCAST(GSum, 1, MPI_DOUBLE_PRECISION, 0,
     $                    MPI_COMM_WORLD, Ierr)

      If (Rank.eq.0) Then
        time2 = MPI_WTime()
        Write (6,*) 'Result=',GSum,' Error=',1-GSum,
     $                ' Time=',time2 - time1
      End If

      Call MPI_FINALIZE(Ierr)

      Stop
      End
```

3.7 Задания

1. На нулевом процессе ввести с клавиатуры значение целого типа в переменную *temp*. Для инициализированных процессов организовать передачу этого значения по кольцу 0→1→2 →…→*size-1* с помощью функций Send и Recv. На процессе с номером *size-1* распечатать значение *temp*.

2. На каждом из *size* инициализированных процессов компоненты массива $y_1, y_2, ..., y_m$ вычисляются по формуле: $y_k = \cos^k(x), k = 1,2,...m, m = 5$, значение x на процессе с номером $rank = 0,1,...size-1$ определяется с помощью датчика случайных чисел: $x = \cos((rank+1)*rand())$. Вычислить максимальный элемент массива $R_0, R_1, ..., R_{size-1}$, $R_i = \prod_{k=1}^{m} y_k, i = 0,1,...size-1$. Выдать результат на нулевом процессе.

3. MPI-процесс с номером 0 вводит с клавиатуры массив из 8 целых чисел. Затем с помощью функции MPI_SCATTER рассылает по 4 процессам фрагменты этого массива. Каждый процесс печатает полученные данные. Произвести сложение всех элементов массива, распределенных по процессам, с помощью функций MPI_REDUCE

с получением результата на процессе с номером 2. Вычисленное значение распечатать.

4. На нулевом процессе с помощью датчика случайных чисел сформировать массив из 10 вещественных чисел: $x_k = \cos(k * rand())\ , k = 1,2,..,10$. С помощью процедуры MPI_BCAST разослать эти значения остальным процессам, умножить на каждом процессе элементы на его номер и определить глобальный максимальный элемент среди всех элементов, содержащихся на каждом процессе.

5. На каждом из *size* инициализированных MPI-процессов компоненты массива $y_1, y_2,..., y_m$ вычисляются по формуле $y_k = e^{\cos(k*x)}$, $k = 1,2,..m, m = 6$, значение x на процессе с номером $rank = 0,1,...size-1$ определяется с помощью датчика случайных чисел: $x = \sin((rank+1) * rand())$. Вычислить минимальное значение среди компонентов массивов y и выдать результат на процессе с номером *size-1*.

6. Процесс с номером *size-1* вводит значение целого типа в переменную *temp*. Рассылает это значение остальным процессам. Процессы с четными номерами меняют знак у *temp* и посылают полученное значение на процессы с нечетными номерами, большими на единицу. Каждый процесс печатает текущее значение переменной *temp*.

7. На каждом из *size* инициализированных процессов элементы массива $y_1, y_2,..., y_m$ вычисляются по формуле $y_k = \sin(kx)/k, k = 1,2,..m, m = 5$, значение x на процессе с номером $rank = 0,1,...size-1$ определяется с помощью датчика случайных чисел: $x = \cos((rank+1) * rand())$. Вычислить сумму всех отрицательных элементов массивов y без использования операций приведения. Результат выдать на процессе с номером 1.

8. На каждом процессе с помощью датчика случайных чисел сгенерировать массив из 5 вещественных чисел: $x_k = \sin(k * (rank+1) * rand()), k = 1,2,...,5$. Распечатать их. Вычислить сумму этих чисел на процессе с номером 1 и получить минимальное число на процессе 2. Отправить полученные результаты на «0»-й процесс и распечатать их. Собрать все значения массивов с

каждого процесса в общем массиве на нулевом процессе. Результат распечатать.

9. Каждый инициализированный процесс с помощью датчика случайных чисел задает переменную $temp = \cos^{rank+1}(rand())$. Найти наибольшее и наименьшее значение *temp* на всех процессах. Распечатать их на нулевом процессе. Заменить на четных процессах *temp<-max(temp)*, а на нечетных *temp<-min(temp)*.

10. На каждом из *size* процессов компоненты массива $y_1, y_2, \ldots, y_m$ вычисляются по формуле $y_k = x^k$, $k = 1,2,\ldots m$, $m = 8$, значение x на процессе с номером $rank = 0,1,\ldots \text{size} - 1$ определяется с помощью датчика случайных чисел: $x = \cos((rank + 1) * rand())$. Вычислить $R_0 + R_1 + \ldots + R_{size-1}$, где $R_i = \sum_{k=1}^{m} \frac{1}{|y_1|+1}$, $i = 0,1,2,\ldots,size-1$, и выдать полученный результат на всех процессах.

11. На нулевом процессе ввести из файла значение целого типа в переменную *temp*. Для инициализированных процессов организовать передачу этого значения по схеме 0→1, 0→2, ... 0→*size*-1 с помощью функции Bcast. Затем каждый процесс умножает *temp* на свой номер, и полученный результат возвращается на нулевой процесс, где он суммируется с результатами, полученными с остальных процессов. Сумма выдается на экран.

12. На каждом процессе определить значение переменной *temp*. Затем с использованием процедур Send и Recv организовать *m*-кратный сдвиг значений *temp* по кольцу 0→1→2→…→*size*-1. Значение *m* определено заранее и известно каждому процессу. Результат сдвига данных распечатать.

13. На каждом процессе определить значение переменной *temp*. Затем с использованием функций Send и Recv организовать передачу значения *temp* от каждого процесса остальным и замену значения *temp* наибольшим. Результат распечатать.

14. Сравнить эффективность передачи данных между двумя процессами без блокировки и с блокировкой.

15. Вычислить скалярное произведение векторов, компоненты которых равномерно распределены между инициализированными процессами. Результат распечатать на «0»-м процессе.

16. Сравнить эффективность коллективной операции Bcast с реализацией процедуры рассылки значения на остальные процессы с помощью функций MPI Send и Recv.

17. Написать MPI-программу, которая считывает из файла или вычисляет по заданной формуле соответствующую часть (блок) одномерного массива ***a*** вещественных чисел размерности ***n*** и заменяет каждый элемент массива (для которого это возможно) на среднее арифметическое соседних элементов. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы.

18. Написать MPI-программу, которая определяет, являются ли линейно независимыми три заданных вектора ***a*** , ***b***, ***c***. Программа должна считывать из файла или вычислять по заданной формуле соответствующую распределенную часть (блок) одномерных массивов ***a*** , ***b***, ***c*** вещественных чисел размерности ***n***. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы.

4 ВЫЧИСЛЕНИЕ КРАТНЫХ ИНТЕГРАЛОВ

Для приближенного вычисления кратных интегралов применяются кубатурные формулы, которые могут быть построены с помощью различных подходов. Наиболее распространены методы повторного применения квадратурных формул и статистических испытаний (метод Монте-Карло).

4.1 Метод Монте-Карло

Вычислим двойной интеграл

$$I = \iint_{\Omega} 3y^2 \sin^2(x)dxdy \ , \ \Omega = \{0 \le x \le \pi, 0 \le y \le \sin(x)\}$$

методом Монте-Карло.

Выберем n случайных точек, равномерно распределенных на $[0,\pi]\times[0,1]$, с координатами $0 \le x_i \le \pi$, $0 \le y_i \le 1$, $i = 1,2,...,n$. Из общего числа n случайных точек in попали в область Ω, остальные n-in оказались вне области. Тогда при достаточно большом n имеет место приближенная формула

$$I \approx \frac{V_\Omega}{in} \sum_{i=1}^{in} f(x_i, y_i), \ f(x,y) = 3y^2 \sin^2(x),$$ здесь объем

$$V_\Omega = \int_0^\pi \sin(x)dx = 2 .$$

4.2 Параллельная программа расчета двойного интеграла методом Монте-Карло

```
      program  Monte_Carlo
      USE IFPORT
C     Вычисление интеграла методом Монте-Карло
      implicit none
      include 'mpif.h'
      double precision s, x, y, f, pi, int, xx, yy, s_total
      double precision in, in_total, ttime, v
      integer size, rank, i, n, err, comm, iseed
C     Задание подынтегральной функции
      f(xx,yy)=3*yy*yy*dsin(xx)**2
```

```
      pi=3.14159265358979d0
      call MPI_INIT(err)
      comm=MPI_COMM_WORLD
      call MPI_COMM_SIZE(comm,size,err)
      call MPI_COMM_RANK(comm,rank,err)
C     Количество реализаций значений последовательности
C     псевдослучайных точек
      n=200000000
      ttime=MPI_WTIME()
C  Генерация «независимой» последовательности случайных
C     величин
      CALL RANDOM_SEED()
      in=0
      s=0
      do i=rank,n,size
        CALL RANDOM_NUMBER(x)
        x=x*pi
        CALL RANDOM_NUMBER(y)
C     Определение количества точек, попавших в область
C     интегрирования
       if (y<=dsin(x)) then
        in=in+1
        s=s+f(x,y)
       endif
      enddo
      call MPI_REDUCE(in, in_total, 1, MPI_DOUBLE_PRECISION,
     &                        MPI_SUM, 0,comm, err)
C     Определяем общую сумму
      call MPI_REDUCE(s, s_total, 1, MPI_double_precision,
     &                          MPI_SUM, 0,comm, err)
C  Вычисление объема V
       v=pi*in_total/n
       ttime=MPI_WTIME()-ttime
       if(rank==0) then
       int=v*s_total/in_total
       write(*,*) 'Количество процессов size = ',size
       write(6,*)' int  = ',int,' time = ',ttime
      endif
      call MPI_FINALIZE(err)
```

```
stop
end
```

Приведем результаты расчета программы **Monte_Carlo** для $n = 200000000$:

```
Количесво процессов size =        1
 int =   1.06641010848839     time =   10.3219280242920

Количесво процессов size =        2
 int =   1.06667979810452     time =   5.20993590354919

Количесво процессов size =        4
 int =   1.06668186722727     time =   2.60214185714722
Количесво процессов size =        8
 int =   1.06650891684341     time =   1.29212307929993

Количесво процессов size =       10
 int =   1.06652993903464     time =   1.04758596420288
```

4.3 Метод повторного применения квадратурных формул

Вычислим двойной интеграл

$$I = \iint_{\Omega} 3y^2 \sin^2(x)dxdy\ ,\ \Omega = \{0 \le x \le \pi, 0 \le y \le \sin(x)\} \qquad (4.1)$$

методом повторного применения квадратурной формулы трапеции.

Идею метода рассмотрим на примере вычисления двойного интеграла $I = \iint_{\Omega} f(x,y)dxdy$, где область $\Omega = \{a \le x \le b, c \le y \le d\}$ представляет собой прямоугольник. В этом случае интеграл принимает вид $I = \int_a^b dx \int_c^d f(x,y)dy$ или

$$I = \int_a^b F(x)dx\,, \qquad (4.2)$$

где $F(x) = \int_c^d f(x,y)dy$. Для вычисления интеграла (4.2) можно при-

менить любую квадратурную формулу, например формулу трапеций.

Положим $h = \frac{b-a}{n}$, $x_i = a + i \cdot h$, $i = 0,1,...,n$ и

$$I = \int_a^b F(x)dx \approx \frac{h}{2}(F_0 + 2F_1 + 2F_2 + \cdots + 2F_{n-1} + F_n),$$

где $F_i = F(x_i) = \int_c^d f(x_i, y)dy$, $i = 0,1,...,n$.

В конечном итоге поставленная задача сводится к вычислению $(n+1)$-интеграла

$$F_i = \int_c^d f(x_i, y)dy.$$

Описанный метод можно применить для криволинейной области.

Для этого запишем интеграл (4.1) в виде

$$I = \int_a^b dx \int_c^d f(x,y)dy, \; a = 0, b = \pi, c = 0, d = \sin x, f(x,y) = 3y^2 \sin^2(x),$$

здесь верхний предел интегрирования d зависит от x.

4.4 Параллельная программа расчета двойного интеграла методом повторного применения квадратурной формулы трапеции

```
      Program povtor
C     Вычисление кратного интеграла с помощью повторного
C     применения квадратурной формулы трапеции

      Implicit None
      Include 'mpif.h'
      Integer Size, Rank, Ierr, I,k, N,m
      Double Precision  GSum,s,s1,eps,df,pi,
     $                  time, a1, b1,hh, h1,a,b,c,d,x,y,f
C     Задание числа интервалов разбиения по x, пределов
C     интегрирования и точности вычисления однократного
C     интеграла
      Parameter (n=100000, pi=3.14159265358979d0, a=0d0, b=pi,
     $           c=0d0, eps=1d-9)
```

```
C
      df(x)=dsin(x)

      Call MPI_INIT(Ierr)
      Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
      Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

C     Каждый процесс определяет «свое» число интервалов
C     разбиения и определяет свои пределы интегрирования по x
      m=n/size
      hh=(b-a)/size
      a1=a+rank*hh
      b1=a1+hh
      h1=(b1-a1)/m
      time=mpi_Wtime()

C     Каждый процесс определяет свою частичную сумму по x
      s=0
      k=0
      if(rank==0) k=1
 !В «общих» граничных точках интеграл F(x) вычисляет
      do i=k,m-1          ! процесс с большим номером rank
       x=a1+i*h1
       d=df(x)
       call integral(c, d, x, eps, s1)
       s=s+s1*h1
      end do

C   Вычисление F(x) в крайних узлах
      if(rank==0) then
       d=df(a1)
       call integral(c, d, a1, eps, s1)
       s=s+s1/2*h1
      end if
      if (rank==size-1) then
       d=df(b1)
       call integral(c, d, b1, eps, s1)
       s=s+s1/2*h1
      end if
```

```
C    Результаты суммирования частичных сумм посылаются на
C    «0» - процесс
     if(size==1) then
        gsum=s
     else
       call mpi_reduce(S, Gsum, 1, mpi_double_precision,
    $                  mpi_sum, 0, mpi_comm_world, ierr)
     end if

      time=mpi_wtime()-time

      if(rank.eq.0) then
       write(6,*) 'Количество ПЭ size = ',size
       write(6,*)' integral =',gsum
        write(6,*)'time = ',time
      end if

       Call MPI_FINALIZE(Ierr)
       Stop
      End

      Double precision function f(x, y)
C     Подынтегральная функция

       Implicit none
       Double precision x,y
       f=3*y*y*dsin(x)**2
       return
       end

      Subroutine integral(c, d, x, eps, s)
C     Вычисление с точностью eps однократного интеграла по y
C     методом трапеций
      Implicit none
      Double precision c, d, s, sum1, sum2, h, f, x, eps

      Integer i, N

      Sum1=(f(x,c)+f(x,d))*0.5*(d-c)
      Sum2=0.5*Sum1+0.5*(d-c)*f(x,0.5*(c+d))
```

```
N=2
h=0.5*(d-c)

Do while(abs(sum1-sum2).gt.eps*3)
 Sum1=sum2

 h=h/2
 N=N*2
 Sum2=Sum1*0.5

 Do i=1,N-1,2
  Sum2=sum2+f(x,c+i*h)*h
 End do
End do
S=sum2
return
end
```

Приведем результаты расчета программы **povtor**:

```
Количество ПЭ size =           1
 integral =   1.06666666825804
time =    6.42147493362427

Количество ПЭ size =           2
 integral =   1.06666666825803
time =    3.21070718765259

Количество ПЭ size =           4
 integral =   1.06666666825802
time =    2.75767612457275

Количество ПЭ size =           8
 integral =   1.06666666825802
time =    1.69904017448425

Количество ПЭ size =          16
 integral =   1.06666666825802
time =   0.851202011108398
```

4.5 Задания

1. Написать MPI-программу вычисления интеграла $\int_a^b f(x)dx$ с точностью ε, используя обобщенную квадратурную формулу трапеций. Для оценки точности использовать правило Рунге. Вычислить ускорение и эффективность параллельной программы.

2. Написать MPI-программу вычисления интеграла $\int_a^b f(x)dx$ с точностью ε, используя обобщенную квадратурную формулу Симпсона. Для оценки точности использовать правило Рунге. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы. Определить ускорение и эффективность параллельной программы.

3. Написать MPI-программу вычисления интеграла $\int_a^b f(x)dx$ с точностью ε, используя обобщенную квадратурную формулу Ньютона («3/8»). Для оценки точности использовать правило Рунге. Вычислить ускорение и эффективность параллельной программы.

Варианты заданий (1–3):

1. $f(x)=\exp(-x^2+0.38)/(2+\sin(1/(1.5+x^2)))$, $a=0.4$, $b=1$.
2. $f(x)=(x^2+\sin(0.48(x+2)))/(\exp(x^2)+0.38)$, $a=0.4$, $b=0.6$.
3. $f(x)=(1-\exp(0.7/x)/(2+x)$, $a=1$, $b=3$.
4. $f(x)=\exp(-\mathrm{tg}(0.84x))/(0.35+\cos(x))$, $a=0$, $b=\pi/2$.
5. $f(x)=\mathrm{arctg}(0.7x)/(x+1.48)$, $a=0.2$, $b=0.5$.
6. $f(x)=\ln(1+x)/x$, $a=0.1$, $b=1$.
7. $f(x)=\exp(-1.46x^2)/(3.5+\sin(x))$, $a=0.3$, $b=0.8$.
8. $f(x)=1/(\sqrt{x}\,(\exp(0.9x)+3))$, $a=0.5$, $b=2$.
9. $f(x)=\sqrt{x(3-x)}\,/(x+1)$, $a=1$, $b=1.2$.
10. $f(x)=\exp(1-x)/(2+\sin(1+x^2))$, $a=0.4$, $b=1$.
11. $f(x)=\sin(x+2)/(0.4+\cos(x))$, $a=-1$, $b=1$.
12. $f(x)=(\sqrt{2+x^2}\,)/((1+\cos 2x)\sqrt{1-x^2}\,)$, $a=0$, $b=1$.
13. $f(x)=(x^2+\sin(0.48(x+2)))/(\exp(x^2)+o,38)$, $a=0.4$, $b=1$.
14. $f(x)=\arcsin(x/0.2)/x$, $a=1$, $b=1.6$.
15. $f(x)=x^2\,\mathrm{arcctg}(x/2)/x$, $a=1$, $b=2$.

16. $f(x)=x/\cos^3(1.4x)$, $a=0.1$, $b=1$.
17. $f(x)=x^4/(0.5x^2+x+6)$, $a=0.4$, $b=1.5$.
18. $f(x)=1/(x(0.2x+1)^{3/2})$, $a=1$, $b=2$.
19. $f(x)=x/\sin^3(2x)$, $a=0.1$, $b=0.5$.

4. Написать MPI-программу вычисления двойного интеграла $\iint_D f(x,y)dxdy$ методом Монте-Карло. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы; вычислить ускорение и эффективность параллельной программы.

5. Написать MPI-программу вычисления двойного интеграла $\iint_D f(x,y)dxdy$ методом повторного применения квадратурной формулы прямоугольников. Протестировать ее на аналитическом решении; вычислить ускорение и эффективность параллельной программы.

6. Написать MPI-программу вычисления двойного интеграла $\iint_D f(x,y)dxdy$ методом повторного применения квадратурной формулы Симпсона. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы; вычислить ускорение и эффективность программы.

7. Написать MPI-программу вычисления двукратного интеграла $\iint_D f(x,y)dxdy$ методом повторного применения квадратурной формулы Ньютона («3/8»). Протестировать ее на аналитическом решении; вычислить ускорение и эффективность программы.

Варианты заданий (4–7):

1. $f(x,y)=x/y^2$, $D=\{0<x<1,2<y<5\}$.
2. $f(x,y)=\exp(x-y)$, $D=\{-1 \le x \le 0,0 \le y \le 1\}$.
3. $f(x,y)=xy$, $D=\{|x+2y| \le 3, |x-y| \le 3\}$.
4. $f(x,y)=|xy(x+y)|$, $D=\{|x|+|y| \le 1\}$.
5. $f(x,y)=(x^2-y^2)^2$, $D=\{|x|<y<1\}$.
6. $f(x,y)=\exp(x+y)^2$, $D=\{0<x<1,0<y<1-x\}$.
7. $f(x,y)=(x^2-y^2)\sin(\pi(x-y)^2)$, $D=\{|y|<x<1-|y|\}$.

8. $f(x,y)=(x^4-y^4)$, $D=\{1<xy<2, 1<x^2-y^2<2, x>0\}$.

9. $f(x,y)=y^2\,(x^2+1)$, $D=\{1<xy<2, 0<x<y<3x\}$.

10. $f(x,y)=(xy)^{-2}$, где область D ограничена прямыми: $y=3x$, $3y=x$, $5x+y=4$, $x+y=4$.

11. $f(x,y)=|\,xy\,|$, $D=\{x^4+y^4<1\}$.

12. $f(x,y)=x+y$, где область D ограничена прямыми: $y^2=2x$, $x+y=4$, $x+y=12$.

13. $f(x,y)=(x+y)^2/x,\ )$, $D=\{1<x+y<3, x<2y<4x\}$.

14. $f(x,y)=x^3+y^3,)$, $D=\{x^2<y<3x^2, 1<2xy<3\}$.

15. $f(x,y)=\cos(x^2+y^2)$, $D=\{x^2+y^2<a^2\}$.

16. $f(x,y)=\ln(1+x^2+y^2)$, $D=\{x^2+y^2<a^2, y>0\}$

17. $f(x,y)=(xy)^2$, $D=\{1<xy<2,\ |\,x-y\,|<1\}$.

18. $f(x,y)=\exp(2x-x^2)$, $D=\{0<y<1, 0<x<1-y\}$.

19. $f(x,y)=xy+y^2$, $D=\{0<x<2, 0<y<2\}$.

20. $f(x,y)=\sqrt{x+y}$, $D=\{0<x<1, 0<y<x\}$.

5 МАТРИЧНЫЕ ВЫЧИСЛЕНИЯ

Задача матричного умножения требует для своего решения выполнения большого количества арифметических операций.

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

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}\ ,\ i, j = 1, ..., n\ . \qquad (5.1)$$

Из (5.1) видно, что для вычисления одного элемента матрицы C необходимо n умножений и n сложений. Учитывая общее количество таких элементов, можно сосчитать, что операция умножения матриц потребует выполнения n^3 скалярных умножений и n^3 сложений на обычном последовательном компьютере:

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

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

5.1 Способы повышения производительности умножения матриц

Производительность умножения матриц может быть улучшена путем изменения порядка циклов ijk в (5.1). Каждый язык программирования характеризуется различным способом хранения в памяти компьютера элементов массивов. На Фортране элементы матриц располагаются последовательно в памяти по столбцам, т.е. $\{a_{11}, a_{21}, ..., a_{n1}, a_{12}, a_{22}, ..., a_{n2}, a_{13}, ..., a_{nn}\}$. Размещаемые в кэш-памяти элементы матриц A, B, C эффективно используются, когда доступ к ним или их модификация в алгоритме умножения матриц осуществляется последовательно с переходом к соседней ячейке памяти. Кроме того, при последовательной выборке данных увеличивается эффективность использования кэш-памяти. Поэтому порядок следо-

вания циклов *jki* при умножении матриц в программе, написанной на Фортране, будет способствовать повышению производительности вычислений по сравнению с *ijk* циклом:

```
 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
```

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

Наоборот, цикл *kij* предпочтительнее использовать при программировании на языках со строковым способом хранения двумерных массивов в оперативной памяти.

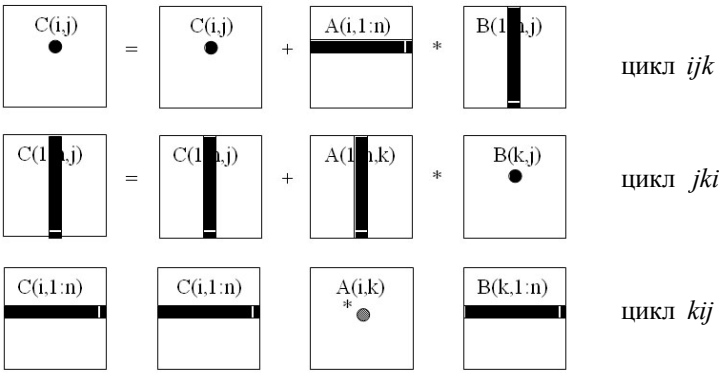

Рис. 5.1 Схема доступа к элементам матриц

В табл. 5.1 представлены результаты оценки временных затрат Фортран-программ, умножающих матрицы размерности: 1000x1000, 2000x2000 и 4000x4000. В программах используются различные способы организации работы вложенных циклов. Расчеты проводились на одном вычислительном узле кластера СКИФ Cyberia. Оценивая полученные данные, можно отметить, что использование циклов *jki* и *kji* имеет преимущество перед другими способами

организации умножения матриц, причем с ростом размерности задачи это преимущество становится значительным.

Таблица 5.1 Время умножения матриц размером 1000x1000, 2000x2000, 4000x4000, с

	ijk	ikj	jik	jki	kij	kji
1000	5,5	11,4	6,5	2,3	12,5	3,8
2000	53,1	110,2	63,4	19,7	123,1	33,1
4000	582,5	1281,5	650,8	163,2	1382,9	260,1

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

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

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}\ , \qquad (5.2)$$

где A_{ik}, B_{kj}, C_{ij} – матрицы размером $(n/N)\times(n/N)$, N – количество блоков по строкам или столбцам.

Чтобы понять, как достигается ускорение, необходимо иметь представление, на что тратится машинное время при выполнении программы и как функционирует память компьютера. Память компьютера имеет иерархическую структуру 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 нижней части (рис. 5.2).

Арифметические и логические операции выполняются только с данными, размещенными в регистрах. С одного уровня иерархической памяти данные могут перемещаться вверх или вниз на соседние уровни, причем скорость передачи данных снижается к нижнему уровню памяти. Скорость обмена данными на нижних уровнях иерархической памяти на порядки может быть ниже скорости, с которой выполняются арифметические и логические операции. 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 матрицы могут разместиться целиком лишь в большом и медленном уровне иерархической памяти.

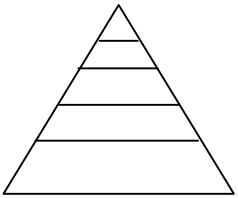

Рис. 5.2 Иерархическая структура памяти компьютера

Рассмотрим, как может повлиять на разработку алгоритма наличие кэш-памяти. Обычно обмен данными между основной памятью и кэш-памятью относительно дорогостоящ. Поэтому, если какие-то данные попали в быструю память, нужно стремиться их использовать максимальным образом. Предположим, что размеры основной 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 способен управлять движением данных.

Рассмотрим неблочную версию матричного умножения (рис. 5.1, 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. 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 (5.2), приводит к более эффективному использованию 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 и записи блоков матрицы C – $N^2(n^2/N^2) = n^2$. Всего получается $\Gamma_2 = 2(N+1)n^2 \approx 2Nn^2$ обращений к быстрой кэш-памяти. Поэтому чтобы минимизировать число обращений, нужно взять как можно меньшее значение N. Но N подчиняется ограничению, что $M \geq (3n^2/N^2)$, которое означает, что в кэш-памяти может 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 $\Gamma_1/\Gamma_2 \approx \sqrt{M}/(2\sqrt{3})$, и видно, что блочные скалярные произведения имеют решительное преимущество. Продемонстрируем это на

основе расчетов на одном вычислительном узле кластера СКИФ Cyberia. Ниже представлен фрагмент программы, реализующей блочное умножение матриц.

```
do j=1,nn
 do k=1,nn
  do i=1,nn
   do i1=1,m
    do j1=1,m
     do k1=1,m
      C(i1, j1, i, j)=C(i1, j1, i, j)+A(i1, k1, i, k)*B(k1, j1, k, j)
    end do
   end do
  end do
 end do
end do
end do
```

Здесь $nn = N$, $m = n / N$.

Таблица 5.2 Время умножения матриц 2000x2000
с использованием блочных алгоритмов, с

	$i1, j1, k1$	$k1, i1, j1$	$j1, k1, i1$
j, k, i	6,2	7,9	9,1
i, j, k	6,3	8,0	8,8
k, i, j	10,0	11,7	12,8

Таблица 5.2 наглядно демонстрирует существенное сокращение времени выполнения операции умножения матриц большой размерности с использованием их блочного представления.

5.2 Распараллеливание операции умножения матриц

Процедура параллельного умножения матриц легко может быть построена на основе алгоритмов скалярного умножения векторов или матрично-векторного умножения для многопроцессорных систем. Введение макроопераций значительно упрощает проблему выбора эффективного способа распараллеливания вычислений, позво-

ляет использовать типовые параллельные методы выполнения макроопераций в качестве конструктивных элементов при разработке параллельных способов решения сложных вычислительных задач.

Для понимания проблемы воспользуемся общей процедурой построения параллельных методов решения задачи матричного умножения:

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

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 произведений.

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

одномерное укрупнение по строкам: в новую подзадачу объединяется $(n/p)\times n\times n$ фундаментальных 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: укрупненная подзадача содержит $n\times(n/p)\times n$ фундаментальных блоков. В этом случае для вычислений необходимы целиком матрица A и матрица B_j, состоящая из $n\times(n/p)$ элементов;

двумерное укрупнение: производится объединение $(n/\sqrt{p})\times(n/\sqrt{p})\times n$ подзадач. Здесь реализуется алгоритм, основанный на блочном представлении матриц: $C_{ij}=\sum_{k=1}^{\sqrt{p}} A_{ik}B_{kj}$,

$i, j = 1, ..., \sqrt{p}$. Подзадача (i, j), полученная после укрупнения, должна 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.

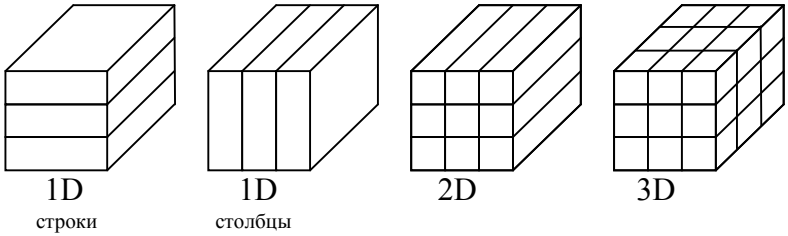

Рис. 5.3 Стратегии укрупнения мелкозернистых подзадач

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

Проведя расчет временных затрат без учета обменных операций между процессорами для каждой из построенных процедур параллельного умножения матриц, получим $T_p = (t_{mult} + t_{add}) n^3 / p$. Тогда

$$S_p = \frac{T_1}{T_p} = \frac{(t_{mult} + t_{add}) n^3}{(t_{mult} + t_{add}) n^3 / p} = p. \qquad (5.3)$$

При рассмотрении параллельной программы, реализующей алгоритм матричного умножения, ограничимся достаточно простым случаем: будем использовать одномерное распределение матрицы произведения по процессам. Это означает, что каждый процесс будет вести расчеты лишь в своей полосе матрицы результата (рис. 5.4). В языке программирования FORTRAN двумерные массивы располагаются в памяти компьютера по столбцам, и это, в свою очередь, означает, что декомпозицию («разрезание» по процессам) матрицы результата следует производить по второму индексу, с той целью, чтобы на каждом процессорном элементе хранились данные из близко расположенных друг к другу ячеек памяти. Это позволит ускорить обращение каждого процесса к элементам своей части массива, что увеличит эффективность программы.

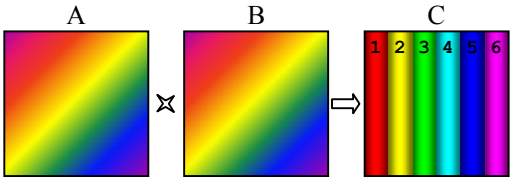

Рис. 5.4 Умножение матриц

Предположим, что нет нужды заботиться об экономии памяти и можно позволить себе хранить перемножаемые матрицы *A* и *B* в памяти каждого процессорного элемента. Тогда не возникает никаких проблем с получением результата: каждый процесс имеет в распоряжении все необходимые данные, чтобы рассчитать свою полосу (блочный столбец) матрицы *C*. После этого необходимо собрать все полученные блоки матрицы *C* с каждого процесса (сделаем это на всех процессах).

```
      Program Example
      Implicit None
      Include 'mpif.h'
      Integer Rank, Size, Ierr, Nm, I, J, K, BCol, ECol, Nr

c     Nm - порядок матрицы
      Parameter (Nm=100)
      Double Precision A(Nm, Nm), B(Nm, Nm), C(Nm, Nm)

      Call MPI_INIT(Ierr)
      Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
      Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)

c     Задаются начальные значения матриц A и B
      If (Rank.eq.0) Then
         Do I = 1,Nm
           Do J = 1,Nm
             A(I,J) = I*J*1.0D0
             B(I,J) = 1/A(I,J)
           End Do
         End Do
       End If
```

```
c  Каждый процесс определяет первый и последний столбец
c  своих расчетов
     BCol = Rank*Nm/Size+1
     ECol = (Rank+1)*Nm/Size

c  «0»-процесс рассылает всем матрицы A и B
     Call MPI_BCAST(A(1,1), Nm*Nm,
    $                       MPI_DOUBLE_PRECISION, 0,
    $                       MPI_COMM_WORLD, Ierr)
     Call MPI_BCAST(B(1,1), Nm*Nm,
    $               MPI_DOUBLE_PRECISION, 0,
    $                       MPI_COMM_WORLD, Ierr)

c  Каждый процесс ведет расчет свой полосы матрицы C
       Do J = BCol, ECol
         Do K = 1,Nm
           Do I = 1,Nm
              C(I,J) = C(I,J)+A(I,K)*B(K,J)
           End Do
         End Do
       End Do
c  Каждый процесс производит рассылку своей полосы
c  остальным процессам
     Do K = 0,Size-1
c  Nr – ширина полосы К-го процесса
       Nr = (K+1)*Nm/Size-K*Nm/Size
     Call MPI_BCAST(C(1,K*Nm/Size+1), Nm*Nr,
    $                MPI_DOUBLE_PRECISION,K,
    $                MPI_COMM_WORLD, Ierr)
     End Do

     if (Rank.eq.0) write(6,*) "A=",A
     if (Rank.eq.0) write(6,*) "B=",B
     if (Rank.eq.0) write(6,*) "C=",C

    Call MPI_FINALIZE(Ierr)

    Stop
    End
```

Стоит обратить внимание на то, каким образом определяется ширина полосы рассылаемых данных: **Nr = (K+1)*Nm/Size-K*Nm/Size**. Так как при вычислении **Nr** используется целочисленное деление, то значение **Nr** для разных процессов, вообще говоря, разное и не равно **Nm/Size**.

Способ улучшения предложенного алгоритма напрашивается сам собой – если в определении полосы матрицы *C* используется лишь такая же полоса матрицы *B*, а не вся матрица целиком, то каждому процессу достаточно послать лишь свою полосу (рис. 5.5).

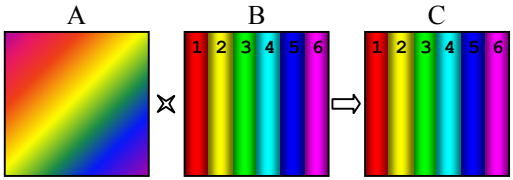

Рис. 5.5 Умножение матриц (улучшенный алгоритм)

Так как ширина рассылаемой полосы различна для разных процессов, то удобнее всего будет воспользоваться векторным вариантом функции распределения блоков данных по всем процессам – функцией **SCATTERV**.

Сбор результатов умножения будет выполняться при вызове функции **MPI_ALLTOALLV**, которая позволяет каждому процессу распределять по всем процессам свои данные и одновременно получать данные других процессов, причем объем данных может быть различным для разных процессов (эта функция является наиболее мощной и гибкой из всего семейства коллективных операций MPI, но одновременно и наиболее сложной в использовании).

```
      Program Example
        Implicit None
        Include 'mpif.h'
        Integer Rank, Size, Ierr, Nm, I, J, K, BCol, ECol, Nr
C     Nm - порядок матрицы
        Parameter(Nm=100)
        Double Precision A(Nm,Nm), B(Nm,Nm), Bc(Nm,Nm),
     $                   C(Nm,Nm), Cc(Nm,Nm)
        Integer Counts(0:Nm), Shifts(0:Nm),
     $          Counts2(0:Nm), Shifts2(0:Nm)
```

```
      Call MPI_INIT(Ierr)
      Call MPI_COMM_SIZE(MPI_COMM_WORLD, Size, Ierr)
      Call MPI_COMM_RANK(MPI_COMM_WORLD, Rank, Ierr)
C     Задаются начальные значения матриц A и B
      If (Rank.eq.0) Then
       Do I = 1,Nm
        Do J = 1,Nm
         A(I,J) = I*J*1D0
         B(I,J) = 1/A(I,J)
        End Do
       End Do
      End If
C     Каждый процесс определяет первый и последний столбец
C     своих расчетов
      BCol = Rank*Nm/Size+1
      ECol = (Rank+1)*Nm/Size
C     Каждый процесс определяет свои массивы размера полос и
C     их положение
      Do I = 0,Size-1
        Counts(I) = ((I+1)*Nm/Size-I*Nm/Size)*Nm
        Shifts(I) = I*Nm/Size*Nm
      End Do
      Do I = 0,Size-1
        Shifts2(I) = Shifts(Rank)
        Counts2(I) = Counts(Rank)
      End Do
C     «0» посылает всем процессам матрицу A и соответствующую
C     полосу матрицы B
      Call MPI_BCAST(A(1,1), Nm*Nm,
     $                 MPI_DOUBLE_PRECISION, 0,
     $                       MPI_COMM_WORLD, Ierr)
      Call MPI_SCATTERV(B(1,1), Counts, Shifts,
     $                     MPI_DOUBLE_PRECISION, Bc(1,1),
     $                     Counts(Rank),MPI_DOUBLE_PRECISION,
     $               0, MPI_COMM_WORLD, Ierr)

      Do J = BCol,ECol
       Do K = 1,Nm
        Do I = 1,Nm
            C(I,J) = C(I,J)+A(I,K)*Bc(K,J-BCol+1)
```

```
      End Do
     End Do
    End Do

C   Каждый процесс отправляет свою рассчитанную полосу C
C   и заполняет недостающие полосы данными, поступающими
C   от других процессов
     Call MPI_ALLTOALLV(C(1,1), Counts2, Shifts2,
    $                   MPI_DOUBLE_PRECISION, Cc(1,1), Counts,
    $                   Shifts, MPI_DOUBLE_PRECISION,
    $                   MPI_COMM_WORLD, Ierr)

     if (Rank.eq.0) write(6,*) "A=",A
     if (Rank.eq.0) write(6,*) "B=",B
     if (Rank.eq.0) write(6,*) "C=",Cc
    Call MPI_FINALIZE(Ierr)
    End
```

Итак, сначала каждый процесс получает матрицу *A* и свой блок матрицы *B*. Таким образом, на каждом процессе в массиве *Bc* (в первых его столбцах) появляются значения, соответствующие выбранному блоку матрицы *B*. Обратите внимание, обращение к матрице *Bc* несколько иное, чем в предыдущем примере к матрице *B*. После проведения вычисления на каждом процессе содержится матрица *C* с соответствующей рассчитанной полосой. После проведения «коллективного обмена информацией» на каждом процессе содержатся идентичные массивы *Cc* с результатом перемножения.

Обсудим подробнее вопросы о том, как подготавливается и производится сложный процесс обмена. В примере используются два массива смещений *Shifts*, *Shifts2* и два массива «размеров» пересылаемых блоков. Содержимое этих массивов для случая использования трех процессов следующее:

Таблица 5.3 Значения массивов смещений и размеров блоков для случая трех процессов

	«0»-процесс	«1»-процесс	«2»-процесс
Counts	3300 3300 3400	3300 3300 3400	3300 3300 3400
Shifts	0 3300 6600	0 3300 6600	0 3300 6600
Counts2	3300 3300 3300	3300 3300 3300	3400 3400 3400
Shifts2	0 0 0	3300 3300 3300	6600 6600 6600

При рассылке функцией **SCATTERV** используются массивы *Counts* и *Shifts*. В соответствии с их содержимым «0»-процессу посылаются 3300 элементов (33 столбца) матрицы *B*, «1»-процессу посылаются 3300 элементов (33 столбца) матрицы *B*, начиная с 3300-го элемента (с первого элемента 34-го столбца), а «2»-процессу посылаются оставшиеся 3400 элементов (последние 34 столбца) матрицы *B*.

При обмене данными MPI_ALLTOALLV каждый «0»-процесс посылает всем по 3300 элементов матрицы *C* (свои рассчитанные 33 столбца) и получает от «1»-процесса 3300 элементов и от «2»-процесса 3400 элементов матрицы *C*. Причем рассылка ведется в таком случае первых 33 столбцов (*Shifts2*), а получение – со сдвигами в соответствии с *Shifts*. Как можно заметить, каждый процесс отправляет всем одни и те же данные и заботится о правильном распределении полученных от других процессов данных.

5.3 Задания

1. Написать программу умножения матриц размером 1500x1500. Оценить время выполнения вычислений для различных вариантов порядка следования вложенных циклов. Дать объяснение полученным результатам.

2. Написать программу умножения блочных матриц размером 1000x1000. Определить размер блоков, при котором достигается минимальное время выполнения программы. Дать объяснение полученным результатам. Провести сравнение с подпрограммой **DGEMM** из пакета BLAS3.

3. Написать MPI-программу, которая считывает из файла или вычисляет по заданной формуле вещественную матрицу $\boldsymbol{A}=\{\boldsymbol{a_{ij}}\}$ $i=1,2,..\ .,\boldsymbol{n}$; $j=1,2,...,\boldsymbol{n}$, рассылает ее по процессорным элементам и определяет номер строки, максимально удаленной от первой строки заданной матрицы. Расстояние между *k*-й и *i*-й строками матрицы $\boldsymbol{A}$ определяется как $\sum_{j=1}^{n} \left| \boldsymbol{a_{kj}} \right| \times \left| \boldsymbol{a_{ij}} \right|$. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы.

4. Написать MPI-программу умножения матрицы $\boldsymbol{A}$ на вектор $\boldsymbol{b}$. Параллельная программа должна считывать из файла или вычислять по заданной формуле вещественную матрицу $\boldsymbol{A}$ размерности $\boldsymbol{n} \times \boldsymbol{n}$ и

вектор $\boldsymbol{b}$ размерности $\boldsymbol{n}$. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы.

5. Написать MPI-программу, которая считывает из файла или вычисляет по заданной формуле вещественную матрицу $\boldsymbol{A}=\{\boldsymbol{a_{ij}}\}$ $i=1,2,...,\boldsymbol{n}$; $j=1,2,...,\boldsymbol{n}$, и определяет норму заданной матрицы $\max\limits_{1\le i\le n}\sum\limits_{j=1}^{n}\left|\ \boldsymbol{a_{ij}}\ \right|$. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы. Исследовать ускорение и эффективность полученной параллельной программы.

6. Написать MPI-программу умножения матрицы $\boldsymbol{A}$ на матрицу $\boldsymbol{B}$. Программа должна считывать из файла или вычислять по заданной формуле соответствующую часть (блок) матриц вещественных чисел $\boldsymbol{A}$ и $\boldsymbol{B}$ размерности $\boldsymbol{n}\times\boldsymbol{n}$. Исследовать ускорение и эффективность полученной параллельной программы.

7. Написать MPI-программу, которая считывает из файла или вычисляет по заданной формуле матрицу $\boldsymbol{A}=\{\boldsymbol{a_{ij}}\}$ $i=1,2,...,\boldsymbol{n}$; $j=1,2,\ldots,\boldsymbol{n}$ вещественных чисел и заменяет матрицу $\boldsymbol{A}$ на матрицу $(\boldsymbol{A}+\boldsymbol{A^T})/2$, где $\boldsymbol{A^T}$ – транспонированная матрица $\boldsymbol{A}$. Обеспечить равномерную загрузку всех процессорных элементов, участвующих в работе программы.

8. Написать MPI-программу, которая производит сглаживание исходной матрицы $\boldsymbol{A}$. Операция сглаживания дает новую матрицу того же размера, каждый элемент которой получается как среднее арифметическое имеющихся соседей соответствующего элемента исходной матрицы. Соседями элемента $\boldsymbol{a_{ij}}$ в матрице назовем элементы $\boldsymbol{a_{kl}}$ $\boldsymbol{i-1\le k\le i+1}$, $\boldsymbol{j-1\le l\le j+1}$, $(\boldsymbol{k},\boldsymbol{l})\ne(\boldsymbol{i},\boldsymbol{j})$. Программа должна считывать из файла или вычислять по заданной формуле соответствующую часть (блок) вещественной матрицы $\boldsymbol{A}=\{\boldsymbol{a_{ij}}\}$ $i=1,2,...,\boldsymbol{n}$; $j=1,2,...,\boldsymbol{n}$ размерности $\boldsymbol{n}\times\boldsymbol{n}$.

9. Написать MPI-программу, которая определяет количество локальных минимумов матрицы $\boldsymbol{A}$ и находит максимум среди всех локальных минимумов. Элемент матрицы $\boldsymbol{a_{ij}}$ называется локальным минимумом, если он строго меньше имеющихся у него соседей.

10. Написать MPI-программу, которая среди строк заданной целочисленной матрицы находит строку с минимальной суммой модулей элементов.

11. Написать MPI-программу, которая среди столбцов заданной матрицы $\boldsymbol{A}$, содержащей элементы $\left|\ \boldsymbol{a_{ij}}\ \right|<5$, находит столбец с минимальным произведением элементов.

12. Для заданной симметричной матрицы, используя степенной метод, найти ее спектральный радиус. Матрица имеет вид

$$\begin{pmatrix} 0 & 2^{-2} & \dots & 2^{-n} \\ 2^{-2} & 0 & \dots & 2^{-n+1} \\ \dots & \dots & \dots & \dots \\ 2^{-n} & 2^{-n+1} & \dots & 0 \end{pmatrix}.$$

13. Написать MPI-программу решения системы линейных уравнений $Ax = b$ методом Якоби, представленным в матрично-векторной форме: $x_{k+1} = D^{-1}(D - A)x_k + D^{-1}b, k = 0,1,2,...$ В матрице A все элементы равны 1 за исключением элементов, расположенных на главной диагонали, которые равны $2n$; n – размерность матрицы. Диагональная матрица $D = \{a_{ii}\}$. Все компоненты вектора b имеют значение $3n-1$.

14. Написать MPI-программу решения системы линейных уравнений $Ax = b$ методом Зейделя в матрично-векторной форме: $x_{k+1} = (D + L)^{-1}(D + L - A)x_k + (D + L)^{-1}b, k = 0,1,2,...$ Здесь D – диагональная матрица, составленная из элементов матрицы A, расположенных на главной диагонали; L – нижняя треугольная матрица, ненулевые элементы которой совпадают с элементами матрицы A, расположенными ниже главной диагонали. В матрице A $a_{ii} = 1$ для нечетных i, $a_{ii} = 2$ для четных i, кроме того, $a_{ii+1} = 0.5$, а все остальные элементы равны нулю; n – размерность матрицы.

6 СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ

Основная задача вычислительной линейной алгебры – это решение систем линейных алгебраических уравнений (СЛАУ):

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\ &\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n. \end{aligned} \tag{6.1}$$

Предполагается, что матрица А неособенная, $\det A \neq 0$, т.е. решение системы (6.1) единственно.

Численные методы решения СЛАУ делятся на две большие группы: прямые и итерационные. Прямые методы при отсутствии ошибок округления за конечное число арифметических операций позволяют получить точное решение $\vec{x}^*$. В итерационных методах задается начальное приближение $\vec{x}^0$ и строится последовательность приближенных решений $\left\{\vec{x}^k\right\} \xrightarrow[k\to\infty]{} \vec{x}^*$, где k – номер итерации. В действительности итерационный процесс прекращается, как только $\vec{x}^k$ становится достаточно близким к $\vec{x}^*$.

Итерационные методы привлекательнее с точки зрения объема вычислений и требуемой памяти, когда решаются системы с матрицами высокой размерности. При небольших порядках системы используют прямые методы либо прямые методы в сочетании с итерационными методами.

6.1 Решение СЛАУ методом Гаусса

Рассмотрим систему линейных алгебраических уравнений

$$Ax = b$$

с невырожденной матрицей A размерностью $n \times n$. Будем считать матрицу A заполненной, т.е. содержащей небольшое число ненулевых элементов.

Одним из прямых методов решения линейных систем (6.1) является применение метода исключения Гаусса. Суть этого метода состоит в том, что матрица A сначала упрощается – приводится эквивалентными преобразованиями к треугольному или диагональному

виду, а затем решается система с упрощенной матрицей. Наиболее известной формой гауссова исключения является та, в которой система 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.

Приведем псевдокод прямого хода метода Гаусса, где значения верхнетреугольной матрицы в процессе вычислений переписываются в соответствующие элементы матрицы A:

```
do k=1,n-1
   do i=k+1,n
     l_ik = a_ik/a_kk
   end do
   do j=k+1,n+1
     do i=k+1,n
        a_ij = a_ij – l_ik*a_kj
     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 системы (6.1) b добавлена к матрице A (столбец $n+1$) и обрабатывается в ходе приведения к треугольному виду.

Рассмотрим параллельный алгоритм метода Гаусса. Для сбалансированной загрузки процессоров исходная матрица коэффициентов A распределяется по p процессорным элементам (ПЭ) циклически, т.е. первая строка расширенной матрицы помещается в 0-й ПЭ, вторая – в 1-й ПЭ, и т.д., p-я – в $(p-1)$-й ПЭ. Затем $(p+1)$-я снова помещается в 0-й ПЭ, $(p+2)$-я – в 1-й ПЭ и т.д. (рис. 6.1).

строки:	**строки:**	…	**строки:**	…	**строки:**
1	*2*		*m*		*p*
p+1	*p+2*		*p+m*		*2p*
…	…	…	…	…	…
(k-1)p+1	*(k-1)p+2*		*(k-1)p+m*		*kp*
ПЭ 0	**ПЭ 1**		**ПЭ *m-1***		**ПЭ *p-1***

Рис. 6.1 Циклическая строчная схема распределения уравнений системы (6.1) по процессорным элементам. $k = n / p$

Строку, которая вычитается из всех остальных строк, после предварительного умножения на нужные коэффициенты, назовем текущей строкой. Общая нумерация строк во всей матрице не совпадает с индексацией строк в каждом ПЭ. В каждом ПЭ индексация строк в массиве начинается с единицы.

Алгоритм прямого хода заключается в следующем. Сначала текущей строкой является строка с индексом 1 в 0-м ПЭ, затем строка с индексом 1 в 1-м ПЭ и т.д., и, наконец, строка с индексом 1 в последнем по номеру ПЭ. После чего цикл по процессорным элементам повторяется и текущей строкой становится строка с индексом 2 в 0-м ПЭ, затем строка с индексом 2 в 1-м ПЭ и т.д. После прямого хода строки матрицы в каждом ПЭ будут иметь вид, показанный на рис. 6.2 (пример приведен для четырех ПЭ, * – вещественные числа).

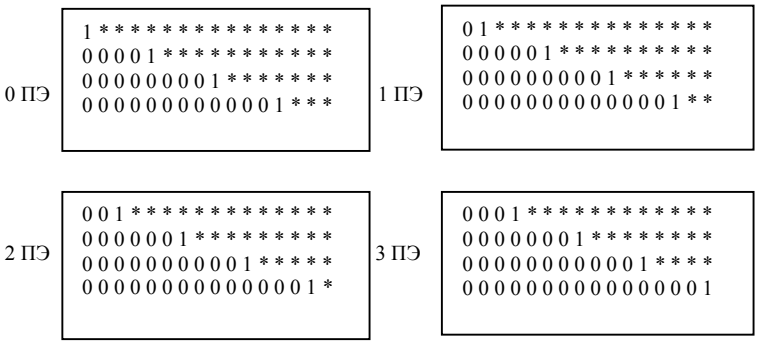

Рис. 6.2 Вид строк матрицы после прямого хода метода Гаусса

Аналогично последовательно по узлам, начиная с последнего по номеру ПЭ, осуществляется обратный ход метода Гаусса.

Особенностью этого алгоритма является то, что как в прямом, так и при обратном ходе ПЭ являются более равномерно загруженными. Процессорные элементы активны в течение всего времени вычислений, и пересылка строк осуществляется всегда на все ПЭ.

Приведем параллельную Фортран-программу численного решения рассмотренным методом системы линейных уравнений с полнозаполненной матрицей, предполагая, что на главной диагонали матрицы в процессе вычислений всегда получаются ненулевые элементы.

6.2 Программа решения СЛАУ методом Гаусса

```
      PROGRAM gauss_method
      Implicit none
      Include 'mpif.h'
      Integer i, j, k, m, n, p, nom
      Integer comm, ierr, size, rank, tag, status(100)
      Parameter (n=1000)
      Double precision a(n,n+1),b(n), x(n), pas(n+1), ch,time

      Call MPI_INIT(ierr)
      Call MPI_comm_size(mpi_comm_world, size, ierr)
      Call MPI_comm_rank(mpi_comm_world, rank, ierr)
      comm = mpi_comm_world

c Каждый ПЭ вычисляет размеры своих полос матрицы A и
c вектора правой части, n – исходная размерность матрицы, m –
c размерность полосы матрицы на ПЭ (размерность матрицы n
c делится без остатка на количество ПЭ)
      m=n/size
c   Задаем исходную матрицу:
c по главной диагонали числа = 2, остальные числа = 1
      do i=1,m
        do j=1,n
          a(i,j)=1.0
          if ( ((i-1)*size+rank+1) .eq. j ) a(i,j)=2.0
        end do
c Столбец свободных членов добавляем (m+1)-й строкой в
c матрицу
         b(i)=n*1.0d0+1
         a(i,n+1)=b(i)
      end do
c Фиксируем время начала расчета
      if (rank.eq.0) time = MPI_Wtime()

c  Прямой ход метода Гаусса
      do k=1,m
        do p=0,size-1
          nom=(k-1)*size+p+1        !!! номер элемента
```

```
с ПЭ с номером rank = p приводит свою строку с номером k
с к диагональному виду. Активная строка k передается всем ПЭ
        if (rank == p) then
         ch=1.0/a(k,nom)
         do j=nom,n+1
          a(k,j)=a(k,j)*ch
         end do
         do j=1,n+1
          pas(j)=a(k,j)
         end do
 !! рассылаем всем ПЭ
         call MPI_Bcast(pas, n+1, MPI_DOUBLE_PRECISION, p,
    $                        comm, ierr)
         do i=m,k+1,-1
          do j=n+1,nom,-1
           a(i,j)=a(i,j)-a(i,nom)*pas(j)
          end do
         end do
        end if
с ПЭ с номером rank < p
        if (rank < p) then
         call MPI_Bcast(pas, n+1, MPI_DOUBLE_PRECISION, p,
    $                        comm, ierr)
         do i=m,k+1,-1
          do j=n+1,nom,-1
           a(i,j)=a(i,j)-a(i,nom)*pas(j)
          end do
         end do
        end if
с ПЭ с номером rank > p
        if (rank > p) then
         call MPI_Bcast(pas, n+1, MPI_DOUBLE_PRECISION, p,
    $                        comm, ierr)
         do i=m,k,-1
          do j=n+1,nom,-1
           a(i,j)=a(i,j)-a(i,nom)*pas(j)
          end do
         end do
        end if
```

```
        end do
      end do
c Конец прямого хода

c Обратный ход
c Циклы по k и p аналогичны, как и при прямом ходе
      do k=m,1,-1
       do p=size-1,0,-1
        nom=(k-1)*size+p+1
         if (rank.eq.p) then
           ch=a(k,n+1)
          call MPI_Bcast(ch, 1, MPI_DOUBLE_PRECISION, p,
     $                comm, ierr)
          do i=k-1,1,-1
            a(i,n+1)=a(i,n+1)-a(k,n+1)*a(i,nom)
          end do
         end if

         if (rank.gt.p) then
          call MPI_Bcast(ch, 1, MPI_DOUBLE_PRECISION, p,
     $                        comm, ierr)
          do i=k-1,1,-1
            a(i,n+1)=a(i,n+1)-ch*a(i,nom)
          end do
         end if

         if (rank.lt.p) then
          call MPI_Bcast(ch, 1, MPI_DOUBLE_PRECISION, p,
     $                        comm, ierr)
          do i=k,1,-1
            a(i,n+1)=a(i,n+1)-ch*a(i,nom)
          end do
         end if
       end do
      end do
c Конец обратного хода
c Вычисляем время работы программы
      if (rank.eq.0) time = MPI_Wtime() - time

c Сбор решения на нулевом процессе
```

```
      do i=1,n
        x(i)=a(i,n+1)
      end do

      call MPI_Gather(x(1), m, MPI_DOUBLE_PRECISION,
     $                x(1), m, MPI_DOUBLE_PRECISION, 0, comm, ierr)
      If (rank.eq.0) then
c Сортировка решения с учетом начального распределения
c  данных
        do i=1,m
          do p=0,size-1
            b((i-1)*size+p+1)=x(i+p*m)
          end do
        end do
c Вывод результата в файл
        open (3,file='rez.txt')
        write (3,'(3f30.25)') time
        do i=1,n
          write (3,'(a,i5,a,f30.20,a)') '  x[',i,']=',b(i),'.'
        end do
      end if
      call MPI_FINALIZE(ierr)
    end
```

Результаты работы программы *gauss_method* для размерности расширенной матрицы $m = 1000$ представлены в табл. 6.1.

Таблица 6.1 Результаты расчетов по программе

Количество ПЭ	1	2	4	8	10
Размер расчетной полосы m	1000	500	250	125	100
Время счета, с	2,28	0,98	0,49	0,27	0,23

6.3 Метод циклической редукции

Рассмотрим систему линейных алгебраических уравнений с трехдиагональной матрицей следующего вида:

$$\begin{aligned} & b_1 \cdot x_1 + c_1 \cdot x_2 = f_1 \ , \\ & a_2 \cdot x_1 + b_2 \cdot x_2 + c_2 \cdot x_3 = f_2 \ , \\ & a_3 \cdot x_2 + b_3 \cdot x_3 + c_3 \cdot x_4 = f_3 \ , \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ & a_{n'} \cdot x_{n'-1} + b_{n'} \cdot x_{n'} = f_{n'} \ . \end{aligned} \qquad (6.2)$$

Системы вида (6.2) 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 на компьютерах с последовательной архитектурой являются специальные варианты метода исключения Гаусса – метод прогонки и метод циклической редукции.

Идея метода циклической редукции заключается в последовательном исключении из уравнений неизвестных x_i сначала с нечетными индексами, а затем с индексами, кратными 2, но не кратными 4, и т.д. Каждый шаг процесса исключения в два раза уменьшает число неизвестных и уравнений, включающих эти неизвестные. На последнем шаге остается одно уравнение, из которого можно найти $x_{\frac{n'}{2}}$. Обратный ход метода состоит в последовательном нахождении неизвестных x_i сначала с номерами *i,* кратными $n'/4$, затем $n'/8$, $n'/16$ и т.д.

Рассмотрим систему линейных алгебраических уравнений (6.2), примем $n' = n-1$, $n = 2^q$, $q > 0$ – натуральное число. Выпишем тройки уравнений системы для $i = 2, 4, \ldots, n-2$:

$$\begin{aligned} & a_{i-1} x_{i-2} + b_{i-1} \cdot x_{i-1} + c_{i-1} \cdot x_i = f_{i-1} \ , \\ & a_i \cdot x_{i-1} + b_i \cdot x_i + c_i \cdot x_{i+1} = f_i \ , \\ & a_{i+1} \cdot x_i + b_{i+1} \cdot x_{i+1} + c_{i+1} \cdot x_{i+2} = f_{i+1} \ . \end{aligned} \qquad (6.3)$$

Если установить $x_0 = x_n = 0$, то получим систему уравнений, эквивалентную (6.2), в которую включены дополнительные уравнения с коэффициентами $a_i = c_i = 0$, $b_i = 1$, $f_i = 0$ для $i = 0$ и $i = n$.

Умножим первое из уравнений (6.3) на α_i, а третье – на γ_i и сложим эти три уравнения, где

$$\alpha_i = -\frac{a_i}{b_{i-1}},\ \gamma_i = -\frac{c_i}{b_{i+1}},\ i = 2,4,...,n-2.$$

При этом x_{i-1}, x_{i+1} исключаются, и в результате от каждой тройки останется по одному уравнению. Совокупность таких уравнений образуют следующую систему:

$$a_i^{(1)} x_{i-2} + b_i^{(1)} \cdot x_i + c_i^{(1)} \cdot x_{i+2} = f_i^{(1)}\ ,\quad i = 2,4,...,n-2,$$

где

$$a_i^{(1)} = \alpha_i \cdot a_{i-1},$$

$$c_i^{(1)} = \gamma_i \cdot c_{i+1},$$

$$b_i^{(1)} = b_i + \alpha_i \cdot c_{i-1} + \gamma_i \cdot a_{i+1},$$

$$f_i^{(1)} = f_i + \alpha_i \cdot f_{i-1} + \gamma_i f_{i+1}\ , i = 2,4,...,n-2.$$

Коэффициенты этой системы уравнений также составляют трехдиагональную матрицу. Следовательно, к ней опять можно применить процесс, описанный выше, до тех пор, пока редукция станет невозможной. При шаге редукции $q-1$ остается только одно уравнение:

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

Так как $x_0 = x_n = 0$, то $x_{\frac{b}{2}} = f_{\frac{n}{2}}^{(q-1)} / b_{\frac{n}{2}}^{(q-1)}$.

Теперь другие неизвестные можно будет найти из процедур замещения. Поскольку нам известны x_0, $x_{\frac{n}{2}}$, x_n, то неизвестные с промежуточными индексами могут быть найдены из уравнений на шаге $q-2$, используя следующее выражение:

$$x_i = (f_i^{(q-2)} - a_i^{(q-2)} \cdot x_{i-\frac{n}{4}} - c_i^{(q-2)} \cdot x_{i+\frac{n}{4}}) / b_i^{(q-2)},\ i = n/4, 3 \cdot n/4.$$

Процедура замещения будет продолжаться до тех пор, пока окончательно не будут найдены все неизвестные. Итак, процедура циклической редукции включает в себя рекурсивное вычисление новых коэффициентов и правых частей для шагов $l = 1,2,...,q-1$ из формул:

$$
\begin{aligned}
a_i^{(l)} &= \alpha_i \cdot a_{i-2^{l-1}}^{(l-1)}, \\
c_i^{(l)} &= \gamma_i \cdot c_{i+2^{l-1}}^{(l-1)}, \\
b_i^{(l)} &= b_i^{(l-1)} + \alpha_i \cdot c_{i-2^{l-1}}^{(l-1)} + \gamma_i \cdot a_{i+2^{l-1}}^{(l-1)}, \\
f_i^{(l)} &= f_i^{(l-1)} + \alpha_i \cdot f_{i-2^{l-1}}^{(l-1)} + \gamma_i \cdot f_{i+2^{l-1}}^{(l-1)},
\end{aligned}
\qquad (6.4)
$$

где $\alpha_i = -a_i^{(l-1)} / b_{i-2^{i-1}}^{(l-1)}$, $\gamma_i = -c_i^{(l-1)} / b_{i+2^{l-1}}^{(l-1)}$ $i = 2^l$ с шагом 2^l до $n - 2^l$ с первоначальными значениями $a_i^{(0)} = a_i$, $b_i^{(0)} = b_i$, $c_i^{(0)} = c_i, f_i^{(0)} = f_i$ и с последующим нахождением неизвестных x_i по формуле

$$
x_i = (f_i^{(l-1)} - a_i^{(l-1)} \cdot x_{i-2^{l-1}} - c_i^{(l-1)} \cdot x_{i+2^{l-1}}) / b_i^{(l-1)}, \qquad (6.5)
$$

где $l = q, q-1, ..., 2, 1$, $i = 2^{l-1}$ с шагом 2^{l-1} до $n - 2^{l-1}$ и $x_0 = x_n = 0$.

На каждом уровне редукции по формулам (6.4) и (6.5) производятся независимые вычисления, которые могут выполняться одновременно. Рассмотрим параллельную реализацию метода. Поскольку число уравнений $n' = 2^q - 1$, для равномерной загрузки число процессорных элементов (ПЭ) будем выбирать как $size = 2^r$, где r, q – натуральные числа $(0 < r < q)$. Каждый ПЭ ведет расчеты в своей части системы из $m = \dfrac{n}{size} + 1$ уравнений и обменивается расчетными данными, которые вычисляются и находятся в адресном пространстве других ПЭ.

Прямой ход редукции состоит из $(q-1)$-го шага. На первом шаге циклическая редукция выполняется для m неизвестных на каждом ПЭ, здесь исключаются неизвестные с нечетными индексами и пересчитываются коэффициенты с индексами i, кратными 2, но не кратными 4, затем коэффициенты $\overline{P}_i^{(1)} = (a_i^{(1)}, b_i^{(1)}, c_i^{(1)}, f_i^{(1)})$ пересылаются: на левый ПЭ – коэффициент с минимальным индексом, кратным 2, но не кратным 4, а на правый ПЭ – с максимальным индексом, кратным 2, но не кратным 4. На 2-м, 3-м,…, $(q-2)$-м шагах пересчитываются и рассылаются по той же схеме коэффициенты $\overline{P}_i^{(l)}$, $l = 2, 3, ...q-2$ с соответствующими индексами i. При прямом ходе максимальная степень параллелизма наблюдается до $(q-r)$-го шага, затем она начинает падать. Это связано с остановкой ПЭ, на которых новые значения коэффициентов уравнений не

пересчитываются. На $(q-1)$-м шаге работает только один ПЭ и вычисляются коэффициенты с индексом 2^{q-1}. Пересылки проводятся на всех уровнях редукции, кроме первого и последнего, каждый ПЭ выполняет 2 двусторонних обмена. На рис. 6.3 представлена диаграмма маршрутизации прямого хода циклической редукции для $n=128$, $size=8$, $m=17$, $q=7$, $r=3$.

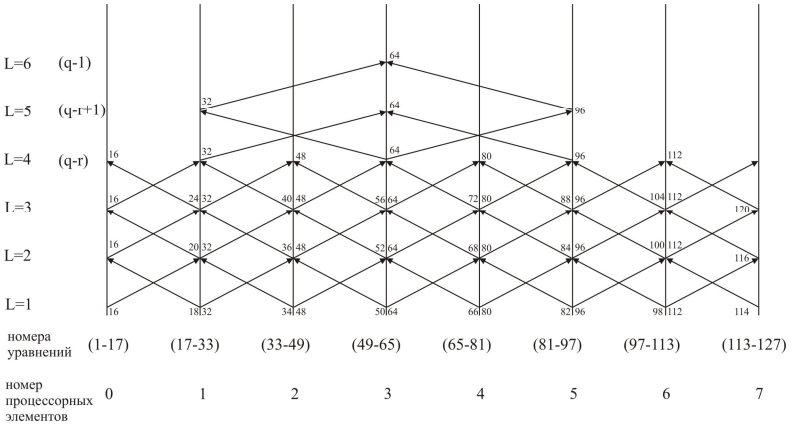

Рис. 6.3 Диаграмма маршрутизации прямого хода циклической редукции

На первом шаге обратного хода ($l=q$) на ПЭ с номером $(size/2-1)$ вычисляется $x_{2^{q-1}}$ и рассылается на 1-й и $(size-2)$-й ПЭ. На втором ($l=q-1$), третьем ($l=q-2$) и т.д. шагах вычисляются неизвестные, кратные 2^{q-2} и не кратные 2^{q-1}, кратные 2^{q-3} и не кратные 2^{q-2}, и т.д. до нахождения всех неизвестных с нечетными номерами. При обратном ходе сначала работает только один ПЭ, затем за r шагов начинают работать остальные. С $(r+1)$-го шага все ПЭ начинают работать автономно без межпроцессорных обменов (рис. 6.4). Пересылки производятся до $(r+1)$-го шага и на последнем шаге.

Таким образом, при прямом ходе наблюдается максимальная степень параллелизма до $(q-r)$-го шага, затем она начинает падать. Это связано с остановкой ПЭ, на которых коэффициенты редукции

не пересчитываются. На $(q-1)$-м шаге остается работать только один процесс. При обратном ходе максимальный параллелизм наблюдается после r-го шага, когда каждый процесс начинает работать без межпроцессорных обменов.

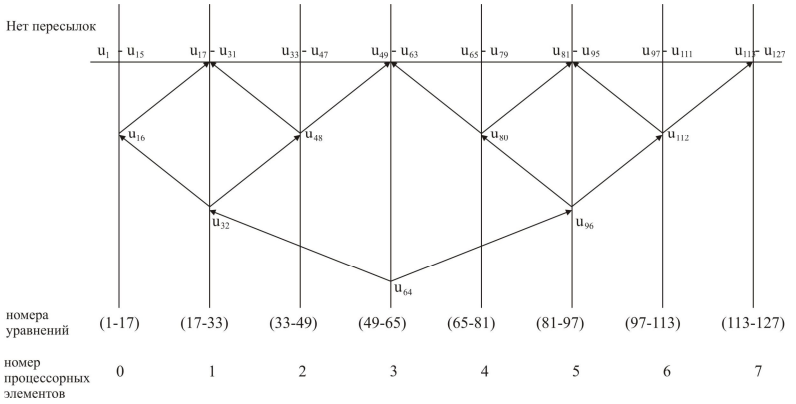

Рис. 6.4 Диаграмма маршрутизации обратного хода метода циклической редукции

6.4 Итерационные методы решения систем линейных уравнений

Итерационные методы для решения линейных систем вида $Ax = b$ с невырожденной квадратной матрицей $A \in R^{n \times n}$ начинают вычислительный процесс с задания начального приближения x_0 и выполняют его последовательное улучшение до тех пор, пока приближенное решение не будет найдено с требуемой точностью. По теории возможно бесконечное число итераций для достижения точного решения. На практике итерации заканчиваются, когда норма невязки $\|b - Ax_k\|$ или другая мера ошибки приближенного решения не станет малой.

Итерационные методы обычно используются для решения систем уравнений, количество которых слишком велико, чтобы их можно было обработать на современном компьютере прямыми методами. Кроме того, эти методы практически незаменимы при решении больших и плохообусловленных систем, поскольку строятся таким образом, чтобы погрешность метода во время поиска реше-

ния 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 получается с меньшими затратами, чем прямыми методами.

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

линейная комбинация векторов (saxpy);

скалярное произведение векторов (dot);

умножение матрицы на вектор (gaxpy);

решение систем с треугольными матрицами.

При параллельной реализации таких операций обычно производится декомпозиция данных и операций по числу используемых параллельных процессов, которая сопровождается передачей данных между процессами для обеспечения локальных вычислений.

Метод Якоби

Получая начальное приближение x_0, в методе Якоби новое приближение к точному решению для каждой компоненты рассчитывается по следующей формуле ($a_{ii} \neq 0$):

$$x_i^{(k+1)} = \frac{b_i - \sum\limits_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)}}{a_{ii}}, i = 1,...,n; k = 0,... . \qquad (6.6)$$

Здесь i – номер компоненты вектора приближенного решения x_k, k – номер итерации.

Если обозначить за D диагональную матрицу, образованную диагональными элементами A, а за L и U – нижнюю и верхнюю треугольные матрицы вида

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

то (6.6) можно записать как

$$x_{k+1} = -D^{-1}\left(L+U\right)x_k + D^{-1}b\,. \qquad (6.7)$$

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 x_k и x_{k+1}, поскольку ни одна из компонент x_k не может быть переписана до тех пор, пока новое приближение x_{k+1} не будет получено. Кроме того, следует обратить внимание, что в (6.6) компоненты следующего приближения x_{k+1} не зависят одна от другой и поэтому они могут быть рассчитаны одновременно, что может быть использовано при создании параллельной реализации метода.

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

$$\left|a_{ii}\right| > \sum_{\substack{j=1 \\ j\neq i}}^{n}\left|a_{ij}\right|, i = 1,...,n\,, \qquad (6.8)$$

что часто наблюдается в большинстве практических задач, то метод сходится, хотя сходимость может быть очень медленной.

В качестве критерия завершения итерационного процесса можно использовать следующее условие:

$$\left\|x_{k+1} - x_k\right\| = \max_i\left|x_i^{(k+1)} - x_i^{(k)}\right| < \varepsilon \;\text{ и }\; \left\|Ax_{k+1} - b\right\| < \varepsilon\,, \qquad (6.9)$$

где x_k – приближенное значение решения на k-м шаге численного метода.

Поскольку вычисления каждой компоненты вектора зависят лишь от значений компонент вектора, рассчитанных на предыдущей итерации, и могут выполняться одновременно, данный метод имеет высокую степень параллелизма. Рассмотрим для параллельной реализации метода Якоби следующий алгоритм. Представим исходную матрицу A в виде блоков из одинакового числа строк расширенной матрицы A. Количество блоков равно числу активированных процессов (рис. 6.5).

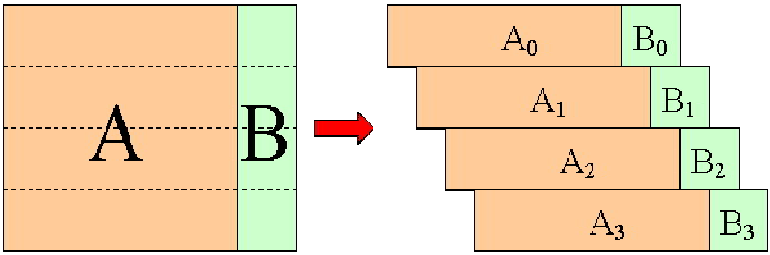

Рис. 6.5 Разбиение исходной матрицы *A* на полосы в случае 4 процессов

После инициализации **MPI**, определения количества активированных процессов в приложении **nprocs**, собственного номера процесса **iam**, заполнения исходной матрицы происходит вычисление параметров, характеризующих расчетную полосу матрицы для каждого процесса. Так, каждый процесс может вычислять фрагмент вектора нового приближенного решения в своей полосе, ограниченной строками расширенной матрицы с номерами **first** и **last**, которые определяются следующим образом:

```
first = iam*m_size/nprocs+1
last = (iam+1)*m_size/nprocs
```

где **iam** – номер процесса; **nprocs** – количество процессов приложения; **m_size** – размерность матрицы.

При этом число значений вектора **xcount**, вычисляемых каждым процессом, будет равно:

```
xcount = (iam+1)*m_size/nprocs - iam*m_size/nprocs
```

Передача фрагмента вектора x_{k+1}, вновь вычисленного процессом, и получение новых значений от других процессов осуществляются при помощи векторной коллективной операции **MPI_ALLTOALLV**. Каждый процесс посылает блок вектора **xx** с числом элементов **xcount2**, удаленных от начала вектора на **xnumer2** значений индекса, всем процессам (в том числе и себе). Полученные от других процессов блоки значений с числом элементов **xcount** записываются в вектор **x,** начиная с позиции, сдвинутой относительно начала вектора на **xnumer**.

```
      call MPI_ALLTOALLV
     $     (xx(1), xcount2, xnumer2, mpi_double_precision,
     $     x(1), xcount, xnumer, mpi_double_precision, comm,
     $     ierr)
```

Использование векторной коллективной операции **MPI_ALLTOALLV** обусловлено тем, что она допускает передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных. Это весьма важно в случае, когда матрица не может быть распределена равномерно на доступных процессорах, т.е. результат деления M_SIZE/NPROCS не является целым числом.

Итерационный процесс продолжается до выполнения условия (6.9). Для его контроля сначала каждый процесс выполняет поиск максимальной по модулю разности **MAX_ERR** между соответствующим новым и старым значениями компонент вектора **X** в своем фрагменте. А затем при помощи глобальной операции редукции среди всех значений **MAX_ERR**, вычисленных процессами, выбирается наибольшее по модулю **MAX_ERR1** и сохраняется в памяти всех процессов.

```
      call MPI_ALLREDUCE (MAX_ERR, MAX_ERR1, 1,
     $     mpi_double_precision, mpi_max, comm, ierr)
```

Аналогично производится вычисление нормы невязки **DRMAX**=$\|Ax_{k+1} - b\|$.

Исходная матрица задается в процедуре **MATGEN**. Формирование элементов на главной диагонали происходит таким образом, чтобы было выполнено условие диагонального преобладания (6.8), необходимое для сходимости, но при этом скорость сходимости была не слишком высокой, что обычно наблюдается в реальных задачах.

6.5 MPI -программа решения СЛАУ методом Якоби

```
      Program Jacobi
      implicit none
      include 'mpif.h'
!
```

```
      integer
     $                nprocs, iam, ierr, i, j,
     $                m_size, first, last, comm, count1
!
      double precision
     $                summ, eps, max_err, max_err1,
     $                time1, dr, drmax
!
      parameter
     $          (m_size = 5000,              ! размер матрицы
     $           eps      = 0.000005)        ! точность вычисления
!
      integer
     $             xcount(0:m_size), xnumer(0:m_size),
     $             xcount2(0:m_size), xnumer2(0:m_size)
!
      double precision
     $                a(m_size, m_size+1),
     $                x(m_size), xx(m_size)
!
       call MPI_INIT (ierr)
       call MPI_COMM_SIZE (mpi_comm_world,nprocs,ierr)
       call MPI_COMM_RANK (mpi_comm_world,iam,ierr)
       comm = mpi_comm_world

! «0»-процесс засекает время и вызывает подпрограмму генерации
! матрицы
       if (iam.eq.0) then
         time1 = mpi_wtime()
         call matgen(a,m_size)
       end if

! каждый процесс определяет первую и последнюю строки своей
! полосы для расчета и соответственно первую и последнюю
! координаты вектора x
       first = iam*m_size/nprocs+1
       last = (iam+1)*m_size/nprocs

! каждый процесс определяет свои массивы размера полос и их
! положение
```

```
      do i = 0, nprocs-1
        xcount (i) = ((i+1)*m_size/nprocs - i*m_size/nprocs)
        xnumer (i) = i*m_size/nprocs
      end do
      do i = 0, nprocs-1
        xcount2 (i) = xcount (iam)
        xnumer2 (i) = xnumer (iam)
      end do
! «0»-процесс рассылает всем процессам матрицу A

      call MPI_BCAST (a(1,1), m_size*(m_size+1),
     $              mpi_double_precision, 0, comm, ierr)

! вычисление начального приближения вектора x и значения
! максимальной ошибки max_err1 всеми процессами
      max_err1=0.0
      do i=1, m_size
        x(i)=a(i,m_size+1)/a(i,i)
        if (x(i).gt.max_err1) max_err1=dabs(x(i))
      end do

! счетчик итераций
      count1 = 1
      drmax=0.0
! основной вычислительный цикл программы
      do while (max_err1.gt.eps.or.drmax.gt.eps)

! вычисление вектора x - нового приближенного решения
! вычисление значения максимальной ошибки max_err и невязки
        max_err=0d0
        do i=first, last
          summ=0
          do j=1, i-1
            summ=summ+a(i,j)*x(j)
          end do
          do j=i+1, m_size
            summ=summ+a(i,j)*x(j)
          end do
          xx(i) = (a(i,m_size+1)-summ)/a(i,i)
          max_err=dmax1(dabs((xx(i)-x(i))),max_err)
```

```
      end do
! вычисление значения максимальной ошибки для всей системы
      call MPI_ALLREDUCE (max_err, max_err1, 1,
     $        mpi_double_precision, mpi_max, comm, ierr)

! рассылка и получение новых значений вектора x
      call MPI_ALLTOALLV
     $        (xx(1), xcount2, xnumer2, mpi_double_precision,
     $     x(1), xcount, xnumer, mpi_double_precision, comm, ierr)

! вычисление значения максимальной невязки drmax всеми
! процессами в своей полосе
      dr=0d0
      do i=first, last
        summ=a(i,m_size+1)
        do j=1, m_size
          summ=summ-a(i,j)*x(j)
        end do
        dr=dmax1(dabs(summ),dr)
      end do

! вычисление значения максимальной невязки для всей системы
      call MPI_ALLREDUCE (dr, drmax, 1,
     $        mpi_double_precision, mpi_max, comm, ierr)
! счетчик итераций
      count1=count1+1

! ограничение на количество итераций
      if (count1.gt.300) goto 11
     end do

! «0»-процесс производит выдачу результатов в файл res.dat
     if (iam.eq.0) then
       time1=MPI_WTIME()-time1
       open (16, file='res.dat')
       write (16,*) 'Vector x  ', 'Absolute error    '
       do i=1,m_size
         summ=0.0
         do j=1,m_size
           summ=summ+x(j)*a(i,j)
```

```
          end do
          write (16,'(2e18.7)') x(i), summ - a(i,m_size+1)
        end do
!
            write (16,'(a, i4)')  'Count of iteration =', count1
            write (16,'(a, e14.6)') 'Norm of residual =',
     $                               drmax
            write (16,'(a, e14.6)')  'Time of calculation =',
     $                                 time1
        write (16,'(a)')          'Correct finish'
        close (16)
       end if
       goto 12

! в случае, если число итераций превысило 300, программа
!завершает работу  и выдает соответствующее сообщение

11   write (6,*)        'Iteration process was diverged'
     write (6,*)        'Max error = ', max_err
12   call MPI_FINALIZE (ierr)
     end
!
! процедура задания исходной матрицы и вектора свободных членов
!
     Subroutine matgen (a, n)
     implicit none
     integer n, i, j
     double precision a(n,n+1)
!
!
      do 30 j = 1,n+1
        do 20 i = 1,n

! задание элементов матрицы
            if(j.le.n) then
              a(i,j) = 1.0d0
            else
              a(i,j)=2.1d0*(n-1)
            end if
20    continue
```

```
30   continue
     do 50 j = 1,n
       a(j,j)=0.0
       do 40 i = 1,n

! задание элементов на главной диагонали матрицы
         if(i.ne.j) a(j,j) = a(j,j) + abs(a(i,j))*1.1d0
40    continue
50   continue
     return
     end
```

6.6 OpenMP-программа решения СЛАУ методом Якоби

Данная программа является записью представленного выше алгоритма с использованием стандарта OpenMP.

```
     Program Jacobi
     implicit none
!
     integer
     $                i, j, m_size, count1
!
     double precision
     $                summ, eps, max_err,
     $                time1, dr, drmax
!
     parameter
     $             (m_size = 5000,          !размер матрицы
     $              eps      = 0.000005)     !точность вычисления
!
     double precision
     $                a(m_size, m_size+1),
     $                x(m_size), xx(m_size)
!
!$   integer       omp_get_num_threads
!$   double precision omp_get_wtime, tm0
!
! фиксируется время начала счета
!$   tm0=omp_get_wtime()
```

```
! отключение выбора количества нитей по умолчанию
!$    call  omp_set_dynamic(.false.) ! if .false. you can change count of
threads
!
! задание числа используемых нитей
!$    call  omp_set_num_threads(4)   ! count of threads = 4
!
! вывод на экран количества используемых нитей
!$omp parallel
!$     write(*,*) 'openmp-parallel with', omp_get_num_threads()
!$omp end parallel
!
! вызов подпрограммы генерации матрицы
      call matgen(a, m_size)
!
! расчет начального приближения
      do i=1, m_size
        x(i)=a(i,m_size+1)/a(i,i)
        if (x(i).gt.max_err) max_err=dabs(x(i))
      end do
!
     drmax=0.0
! счетчик итераций
      count1 = 1

! основной вычислительный цикл программы
      do while (max_err.gt.eps.or.drmax.gt.eps)
        max_err=0d0
        drmax=0d0
!$omp parallel
!$omp do private (summ) reduction(max:max_err)
        do i=1, m_size
          summ=0
          do j=1, i-1
            summ=summ+a(i,j)*x(j)
          end do
          do j=i+1, m_size
            summ=summ+a(i,j)*x(j)
          end do
          xx(i) = (a(i,m_size+1)-summ)/a(i,i)
```

```
            summ=dabs((xx(i)-x(i)))
            max_err=dmax1(max_err,summ)
        end do
!$omp end do
!$omp do private (summ) reduction(max:drmax)
        do i=1, m_size
          x(i) = xx(i)
          summ=a(i,m_size+1)
          do j=1,m_size
            summ=summ-a(i,j)*x(j)
          end do
          drmax=dmax1(drmax,dabs(summ))
        end do
!$omp end do
!$omp end parallel
!
        count1=count1+1
        if (count1.gt.300) goto 11
      end do
!$     tm0=omp_get_wtime()-tm0
!
! проверка
      open(16, file='res.dat')
      write(16,*) 'Vector x     ', 'Absolute error    '
      do i=1,m_size
          summ=0.0
          do j=1,m_size
            summ=summ+x(j)*a(i,j)
          end do
          write(16,'(2e18.7)') x(i), summ - a(i,m_size+1)
      end do
!
      write(6,'(a, i4)')      'Count of iteration = ', count1
      write(6,'(a, e14.6)')      'Residual norm =', drmax
!
! фиксируется время окончания счета и выдается время работы
! программы
!$     write(6,*)       'Time of calculation =', tm0
      write(6,'(a)')     'Correct finish'
      close(16)
```

```
! в случае, если число итераций превысило 300, программа
! завершает работу и выдает соответствующее сообщение
      goto 12
11    write (6,*) 'Iteration pricess was diverged'
      write (6,*) 'Maximum error =', max_err
12  end
!
!
     Subroutine matgen (a, n)
     implicit none
     integer n, i, j
     double precision a(n,n+1)
!
!
      do 30 j = 1,n+1
        do 20 i = 1,n

! задание элементов матрицы
             if(j.le.n) then
                a(i,j) = 1.0d0
             else
                a(i,j)=2.1d0*(n-1)
             end if
20    continue
30    continue
      do 50 j = 1,n
        a(j,j)=0.0
        do 40 i = 1,n

! задание элементов на главной диагонали матрицы
          if(i.ne.j) a(j,j) = a(j,j) + abs(a(i,j))*1.1d0
40    continue
50    continue
     return
     end
!
```

6.7 Оценка ускорения параллельной программы решения СЛАУ методом Якоби

Оценка ускорения параллельной программы решения СЛАУ методом Якоби проводилась на матрице размерами 5000x5000 с точностью вычислений EPS=0.000005. Итерационный процесс сошелся за 226 итераций. Зависимость времени счета от количества активированных процессов, а также ускорение параллельной MPI-программы решения СЛАУ методом Якоби представлены в табл. 6.2.

Из результатов следует, что ускорение растет с ростом числа используемых процессоров. Причем наблюдается существенное сокращение времени счета – более чем в 20 раз на 50 процессорах. Однако эффективность распараллеливания метода Якоби падает при увеличении числа задействованных процессорных элементов. Это обусловлено тем, что с ростом доступных процессоров уменьшается количество уравнений системы, обрабатываемых одним процессом, и в то же время увеличивается доля обменов в суммарных временных затратах параллельной программы. Поэтому запуск данной программы на большом количестве процессоров дает невысокую эффективность и имеет смысл только в случае, когда время счета является критическим параметром.

Таблица 6.2 Время счета и ускорение параллельной MPI-программы

	Количество процессоров					
	1	2	5	10	20	50
Время счета, с	117	60	32	17	10	5,4
Ускорение	1,00	1,95	3,66	6,89	11,70	21,67
Эффективность		0,98	0,73	0,69	0,59	0,43

Сравнение времени выполнения и ускорения параллельных программ для решения СЛАУ методом Якоби, написанных с использованием MPI и OpenMP, представлено в табл. 6.3. Расчеты выполнены на кластере ТГУ СКИФ Cyberia. Размер матрицы 5000x5000.

Таблица 6.3 Сравнение времени выполнения и ускорения параллельных программ для решения СЛАУ методом Якоби

		1	2	4
MPI	Время, с	117	60	39
	Ускорение	1,00	1,95	3,00
OpenMP	Время, с	117	60	37
	Ускорение	1,00	1,95	3,16

Таким образом, использование MPI и OpenMP позволяет одинаково ускорить вычисления. Однако трудоемкость написания программы под OpenMP на порядок меньше. Существенным плюсом является возможность использования OpenMP-программ на широко распространенных сейчас персональных компьютерах и серверах с многоядерной архитектурой. В то же время MPI имеет преимущество, когда число ядер в процессорах относительно невелико, поскольку позволяет проводить вычисления на нескольких десятках процессоров и достигать существенного ускорения.

6.8 Задания

1. Используя представленную выше параллельную программу, решить методом Гаусса систему линейных уравнений вида:

$$\begin{cases} 2nx_1 + x_2 + \ldots + x_{n-1} + x_n = 2n, \\ x_1 + 2nx_2 + \ldots + x_{n-1} + x_n = 2n, \\ \qquad\qquad \ldots \\ x_1 + x_2 + \ldots + 2nx_{n-1} + x_n = 2n, \\ x_1 + x_2 + \ldots + x_{n-1} + 2nx_n = n-1. \end{cases}$$

Исследовать ускорение параллельной программы для различной размерности задачи и различного числа используемых процессов.

2. Используя любую представленную выше параллельную программу, решить систему линейных уравнений вида:

$$\begin{cases} 4x_1 + x_2 = 9, \\ x_1 + 4x_2 + x_3 = 7, \\ x_2 + 4x_3 + x_4 = 6, \\ x_3 + 4x_4 + x_5 = 6, \\ \ldots \\ x_{n-2} + 4x_{n-1} + x_n = 6, \\ x_{n-1} + 4x_n = 5. \end{cases}$$

Исследовать ускорение параллельной программы для различной размерности задачи и различного числа используемых процессов.

3. Используя любую представленную выше параллельную программу, составить MPI-программу обращения трехдиагональных матриц. Применить ее для определения обратной матрицы к матрице

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \ldots & 0 & 0 \\ -1 & 2 & -1 & 0 & \ldots & 0 & 0 \\ 0 & -1 & 2 & -1 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & 0 & \ldots & 2 & -1 \\ 0 & 0 & 0 & 0 & \ldots & -1 & 2 \end{pmatrix}.$$

4. Используя разработанную пользователем программу решения трехдиагональных систем методом прогонки, написать параллельную программу обращения трехдиагональных матриц, в которой каждый процесс решает n/p систем, где p – число процессов, n – размер квадратной матрицы. Исследовать ускорение параллельной программы для различной размерности задачи и различного числа используемых процессов.

5. Написать MPI-программу решения системы линейных уравнений $Ax = b$ методом Якоби. В матрице A все элементы равны 1 за исключением элементов, расположенных на главной диагонали, которые равны $2n$; n – размер матрицы. Все компоненты вектора b имеют значение $3n-1$. Исследовать ускорение параллельной программы для различной размерности задачи и различного числа используемых процессов.

6. Написать MPI-программу решения системы линейных уравнений $Ax = b$ методом Якоби. В матрице A $a_{ii} = 1$ для нечетных

i, $a_{ii} = 2$ для четных i, кроме того, $a_{ii+1} = 0{,}5$, а все остальные элементы равны нулю; n – размерность матрицы. Исследовать ускорение параллельной программы для различной размерности задачи и различного числа используемых процессов.

7. Дана система линейных алгебраических уравнений вида $Ax = b$, где A – верхнетреугольная матрица размером $N \times N$ ($N = 2000$), коэффициенты которой рассчитываются по следующей формуле:

a(i,j)=0 если *i*>*j*,

a(i,j)=50 если *i*=*j*,

a(i,j)=0.01 если *i*<*j*.

Столбец свободных членов имеет вид *b(i)*=*i*. Выяснить, можно ли решить эту систему методом Гаусса–Зейделя, в случае положительного ответа составить параллельную программу, решить систему, сделать проверку, экспериментально исследовать ускорение параллельной программы. Метод Гаусса–Зейделя имеет вид:

$$x_i^{k+1} = \frac{1}{a_{ij}}\left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right); \qquad k = 0,1,2,...; \qquad i = 1,n.$$

8. Разработать параллельный алгоритм, написать и отладить параллельную программу решения СЛАУ методом сопряженных градиентов (см. п. 9). Решить систему алгебраических уравнений вида *Ax*=*b*, где *A* – матрица размерностью $N \times N$ ($N = 1000$), коэффициенты которой рассчитываются по следующей формуле:

a(i,j)=*i*+*j*, если $i \neq j$,

a(i,j)=*i*+*j*+$\sum_{j=1}^{N} a_{ij}$, если *i*=*j*,

b(i)= i^2.

Сделать проверку, исследовать ускорение параллельной программы.

9. Используя представленную выше параллельную программу, решить методом Гаусса систему линейных уравнений вида:

$$\begin{cases} 2nx_1 - x_2 - \ldots - x_{n-1} - x_n = n+1, \\ -x_1 + 2nx_2 - \ldots - x_{n-1} - x_n = n+1, \\ \ldots \\ -x_1 - x_2 - \ldots + 2nx_{n-1} - x_n = n+1, \\ -x_1 - x_2 - \ldots - x_{n-1} + 2nx_n = n+1. \end{cases}$$

Исследовать ускорение параллельной программы для различной размерности задачи и различного числа используемых процессов.

10. Используя представленную выше параллельную программу, решить методом Якоби систему линейных уравнений вида:

$$\begin{cases} nx_1 + x_2 + \ldots + x_{n-1} + x_n = 2n+8, \\ x_1 + nx_2 + \ldots + x_{n-1} + x_n = 2n+8, \\ \ldots \\ x_1 + x_2 + \ldots + nx_{n-1} + x_n = 2n+8, \\ x_1 + x_2 + \ldots + x_{n-1} + nx_n = 11n-1. \end{cases}$$

Исследовать ускорение параллельной программы для различной размерности задачи и различного числа используемых процессов.

7 ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ РЕШЕНИЯ ЗАДАЧИ КОШИ ДЛЯ СИСТЕМЫ ОДУ

Моделирование поведения сложных динамических систем, протекания кинетических химических реакций требует решения систем обыкновенных дифференциальных уравнений (ОДУ) высокого порядка. Сокращение времени решения и, следовательно, времени моделирования возможно при использовании мультипроцессорных систем, эффективность работы которых существенно зависит от характеристик применяемых параллельных алгоритмов.

7.1 Постановка задачи

Будем рассматривать систему линейных неоднородных обыкновенных дифференциальных уравнений с постоянными коэффициентами

$$\frac{d\vec{y}}{dt} = A\vec{y} + \vec{f}(t) \tag{7.1}$$

со следующими начальными условиями:

$$\vec{y}(t_0) = \vec{y}_0 = \begin{pmatrix} y_{10} \\ y_{20} \\ \cdots \\ y_{n0} \end{pmatrix}. \tag{7.2}$$

Здесь $\vec{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \cdots \\ y_n(t) \end{pmatrix}$ и $\vec{f}(t) = \begin{pmatrix} f_1(t) \\ f_2(t) \\ \cdots \\ f_n(t) \end{pmatrix}$, а A – матрица с постоянными коэффициентами размером $n \times n$. Решение такой системы ищется обычно с использованием одношаговых или многошаговых методов четвертого порядка.

7.2 Одношаговые методы Рунге–Кутты четвертого порядка

Для построения вычислительных схем методов Рунге–Кутты четвертого порядка в тейлоровском разложении искомого решения $\vec{y}(t)$ учитываются члены, содержащие степени шага h до четвертой

включительно. После аппроксимации производных правой части ОДУ $A\vec{y} + \vec{f}(t)$ получено семейство схем Рунге–Кутты четвертого порядка, из которых наиболее используемой в вычислительной практике является следующая:

$$\vec{y}(t+h) = \vec{y}(t) + (\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)/6 + \vec{O}(h^5), \qquad (7.3)$$

где

$$\vec{k}_1 = hA\vec{y}(t) + h\vec{f}(t),$$
$$\vec{k}_2 = hA(\vec{y}(t) + \vec{k}_1/2) + h\vec{f}(t + h/2),$$
$$\vec{k}_3 = hA(\vec{y}(t) + \vec{k}_2/2) + h\vec{f}(t + h/2),$$
$$\vec{k}_4 = hA(\vec{y}(t) + \vec{k}_3) + h\vec{f}(t + h).$$

Схема (7.3) на каждом шаге h требует вычисления правой части системы ОДУ в четырех точках: $\vec{k}_1, \vec{k}_2, \vec{k}_3, \vec{k}_4$. По формулам (7.3) видно, что расчет значений компонент этих векторов должен выполняться последовательно. Кроме того, следует отметить, что при вычислении каждого вектора $\vec{k}_i$ выполняется операция *gaxpy* – умножение матрицы на вектор и суммирование с другим вектором.

7.3 Параллельная реализация метода Рунге–Кутты четвертого порядка

Поскольку в описанной выше вычислительной схеме наиболее трудоемкой является операция умножения матрицы на вектор при вычислении $\vec{k}_i$, $i = 1,2,3,4$, то основное внимание будет уделено распараллеливанию этой операции. Здесь будет применяться алгоритм скалярных произведений при умножении матрицы на вектор. Поэтому для инициализации будем использовать следующую схему декомпозиции данных по имеющимся процессорным элементам (ПЭ) с локальной памятью: на каждый μ-й ПЭ ($\mu = 0,..., p-1$) распределяется блок строк матрицы $[A]_\mu = A(1 + \mu n/p : (\mu+1)n/p, 1:n)$, вектор $\vec{y}_0$. Принимаем $m = 0$. m – номер вычислительного уровня. Далее расчеты производятся по следующей схеме:

1) на каждом ПЭ с идеальным параллелизмом вычисляются соответствующие компоненты вектора $\vec{k}_1$ по формуле $\left[\vec{k}_1\right]_\mu = h\left(\left[A\right]_\mu \vec{y}_m + \left[\vec{f}(t_m)\right]_\mu\right)$;

2) для обеспечения второго расчетного шага необходимо провести сборку вектора $\vec{k}_1$ целиком на каждом ПЭ. Затем независимо выполняется вычисление компонент вектора $\vec{k}_2$ по формуле $\left[\vec{k}_2\right]_\mu = h\left(\left[A\right]_\mu (\vec{y}_m + \vec{k}_1/2) + \left[\vec{f}(t_m + h/2)\right]_\mu\right)$;

3) проводится сборка вектора $\vec{k}_2$ на каждом ПЭ, вычисляются компоненты вектора $\vec{k}_3$:

$\left[\vec{k}_3\right]_\mu = h\left(\left[A\right]_\mu (\vec{y}_m + \vec{k}_2/2) + \left[\vec{f}(t_m + h/2)\right]_\mu\right)$;

4) проводится сборка вектора $\vec{k}_3$ на каждом ПЭ, вычисляются компоненты вектора $\vec{k}_4$: $\left[\vec{k}_4\right]_\mu = h\left(\left[A\right]_\mu (\vec{y}_m + \vec{k}_3) + \left[\vec{f}(t_m + h)\right]_\mu\right)$;

5) на заключительном шаге каждой итерации метода рассчитываются с идеальным параллелизмом компоненты вектора $\vec{y}_{m+1}$:

$$\left[\vec{y}_{m+1}\right]_\mu = \left[\vec{y}_m\right]_\mu + \left(\left[\vec{k}_1\right]_\mu + 2\left[\vec{k}_2\right]_\mu + 2\left[\vec{k}_3\right]_\mu + \left[\vec{k}_4\right]_\mu\right)/6$$

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 умножения матрицы на вектор (порядка $O(n^2/p)$ арифметических операций), восемнадцать операций сложения векторов и умножения вектора на число (порядка $O(n/p)$ арифметических операций) и четыре операции глобальной сборки векторов.

Рассмотрим алгоритм, который имеет меньшее число межпроцессорных пересылок данных на одной итерации метода Рунге–Кутты. Для этого получим полный оператор для одного вычислительного шага, позволяющий определить значение вектора неиз-

вестных на следующей итерации $\vec{y}_{m+1}$ через $\vec{y}_m$. Для удобства проведения выкладок положим $\vec{f}(t) \equiv 0$. Тогда можно записать, что

$$\vec{k}_1 = hA\vec{y}_m,$$

$$\vec{k}_2 = hA(\vec{y}_m + \vec{k}_1/2) = \vec{k}_1 + (h/2)A\vec{k}_1 = (E + (h/2)A)\vec{k}_1 =$$

$$= h(E + (h/2)A)A\vec{y}_m,$$

$$\vec{k}_3 = hA(\vec{y}_m + \vec{k}_2/2) = \vec{k}_1 + (h/2)A\vec{k}_2 = \vec{k}_1 + (h/2)A(\vec{k}_1 + (h/2)A\vec{k}_1) =$$

$$= h(E + (h/2)A + (h/2)^2 A^2)A\vec{y}_m,$$

$$\vec{k}_4 = hA(\vec{y}_m + \vec{k}_3) = \vec{k}_1 + (hA + (h^2/2)A^2 + (h^3/4)A^3)\vec{k}_1 =$$

$$= h(E + hA + (h^2/2)A^2 + (h^3/4)A^3)A\vec{y}_m,$$

и, собрав все, в итоге получим

$$\vec{y}_{m+1} = \left\{E + hA(E + (h/2)A(E + (h/3)A(E + (h/4)A)))\right\}\vec{y}_m. \qquad (7.4)$$

Рассчитав заранее матрицу оператора, заключенного в фигурные скобки в (7.4), получим вычислительный алгоритм, в котором каждое новое значение вектора $\vec{y}_{m+1}$ рассчитывается за один шаг.

При параллельной реализации правила (7.4) умножение блочных строк матрицы $\tilde{A} = \left\{E + hA(E + (h/2)A(E + (h/3)A(E + (h/4)A)))\right\}$ на вектор $\vec{y}_m$ будет проводиться независимо с идеальным параллелизмом. Потребуется только после каждой глобальной итерации расчетов по правилу (7.4) проводить сборку вектора $\vec{y}_{m+1}$ на всех процессорных элементах.

7.4 Многошаговые методы Адамса. Схема «предиктор–корректор»

При решении задачи Коши методами Рунге–Кутты необходимо вычислять правые части обыкновенных дифференциальных уравнений (7.1) в нескольких промежуточных точках отрезка интегрирования. Количество таких вычислений зависит от порядка используемого метода. Заметим, однако, 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 для получения вектора решения $\vec{y}_{q+1}$. Ме-

тоды 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 формулу Адамса – Башфорта

$$\vec{y}_{m+1} = \vec{y}_m + (h/24)[55(A\vec{y}_m + \vec{f}_m) - 59(A\vec{y}_{m-1} + \vec{f}_{m-1}) + \\ + 37(A\vec{y}_{m-2} + \vec{f}_{m-2}) - 9(A\vec{y}_{m-3} + \vec{f}_{m-3})], \quad (7.5)$$

которая имеет пятый порядок локальной погрешности и четвертый – глобальной. В (7.5) $\vec{f}_{m-i} = \vec{f}(t_m - ih), i = -1,...,3$.

Также приведем обладающую подобными параметрами погрешности интерполяционную формулу Адамса – Моултона

$$\vec{y}_{m+1} = \vec{y}_m + (h/24)[9(A\vec{y}_{m+1} + \vec{f}_{m+1}) + 19(A\vec{y}_m + \vec{f}_m) - \\ - 5(A\vec{y}_{m-1} + \vec{f}_{m-1}) + A\vec{y}_{m-2} + \vec{f}_{m-2})]. \quad (7.6)$$

Последняя формула является неявной, так как искомая величина $\vec{y}_{m+1}$ используется для вычисления значения выражения $A\vec{y}_{m+1}$, которое входит в правую часть. Выражение (7.6) можно рассматривать как уравнение относительно неизвестной величины $\vec{y}_{m+1}$. Для решения такого уравнения наиболее часто используется итерационный метод последовательных приближений, хотя может быть применен и прямой метод решения линейных систем. Для сокращения вычислительных затрат решение, определенное по экстраполяционной формуле (7.5), обычно выбирается в качестве начального приближения для итерационных методов. Часто (7.5) рассматривается как формула прогноза, т.е.

$$\tilde{\vec{y}}_{m+1} = \vec{y}_m + (h/24)[55(A\vec{y}_m + \vec{f}_m) - 59(A\vec{y}_{m-1} + \vec{f}_{m-1}) + \\ + 37(A\vec{y}_{m-2} + \vec{f}_{m-2}) - 9(A\vec{y}_{m-3} + \vec{f}_{m-3})], \quad (7.7)$$

а формула (7.6)

$$\vec{y}_{m+1} = \vec{y}_m + (h/24)[9(A\tilde{\vec{y}}_{m+1} + \tilde{\vec{f}}_{m+1}) + 19(A\vec{y}_m + \vec{f}_m) - \\ - 5(A\vec{y}_{m-1} + \vec{f}_{m-1}) + A\vec{y}_{m-2} + \vec{f}_{m-2})] \quad (7.8)$$

является формулой коррекции, поскольку она имеет меньшую локальную погрешность вычислений по сравнению с (7.5). Последовательное применение (7.7) и (7.8) носит название схемы «предиктор–корректор».

7.5 Параллельный алгоритм многошагового метода Адамса. Схема «предиктор–корректор»

Численное решение задачи Коши для системы ОДУ с постоянными коэффициентами (7.1) можно получить последовательно по шагам с помощью формул четвертого порядка точности Адамса – Башфорта (7.5) и Адамса – Моултона (7.6). Сначала по формуле Адамса – Башфорта (7.7) вычисляются значения, которые являются прогнозом приближенного решения. Затем эти величины используются для расчета скорректированных значений, вычисляемых по формуле Адамса – Моултона (7.8). Необходимые для корректного начала расчетов по формулам (7.7) и (7.8) стартовые значения вычисляются по методу Рунге – Кутты четвертого порядка точности.

Рассмотрим параллельную реализацию схемы «предиктор–корректор».

1. *Декомпозиция*.

Пусть μ -й процессорный элемент ($\mu = 0,..., p-1$) решает n/p уравнений системы, где p – количество процессоров.

0 ПЭ	1 ПЭ	…	*p-1* ПЭ
$[A]_0, [\vec{y}_m]_0, [\vec{y}_{m-1}]_0,$ $[\vec{y}_{m-2}]_0, [\vec{y}_{m-3}]_0$	$[A]_1, [\vec{y}_m]_1, [\vec{y}_{m-1}]_1,$ $[\vec{y}_{m-2}]_1, [\vec{y}_{m-3}]_1$		$[A]_{p-1}, [\vec{y}_m]_{p-1}, [\vec{y}_{m-1}]_{p-1},$ $[\vec{y}_{m-2}]_{p-1}, [\vec{y}_{m-3}]_{p-1}$

Здесь $[A]_\mu$ – блок матрицы A, состоящий из n/p строк и n столбцов. $[\vec{y}]$ – часть вектора $\vec{y}$, включающая n/p компонентов.

2. *Параллельный алгоритм* (необходимые для обеспечения корректности начала вычислений по многошаговым методам Адамса значения векторов $\vec{y}_0, \vec{y}_1, \vec{y}_2, \vec{y}_3$ рассчитываются на одном процессо-

ре методом Рунге–Кутты четвертого порядка и передаются на остальные ПЭ):

а) инициализация ($m = 3$):

на каждом μ -м ПЭ вычисляем $\left[\vec{V}^0\right]_\mu = [A]_\mu \cdot \vec{y}_{m-3} + \left[\vec{f}_{m-3}\right]_\mu$,

$\left[\vec{V}^1\right]_\mu = [A]_\mu \cdot \vec{y}_{m-2} + \left[\vec{f}_{m-2}\right]_\mu$, $\left[\vec{V}^2\right]_\mu = [A]_\mu \cdot \vec{y}_{m-1} + \left[\vec{f}_{m-1}\right]_\mu$,

$\left[\vec{V}^3\right]_\mu = [A]_\mu \cdot \vec{y}_m + \left[\vec{f}_m\right]_\mu$;

б) на каждом μ -м ПЭ выполняем шаг «предиктор»

$$\left[\tilde{\vec{y}}_{m+1}\right]_\nu = [\vec{y}_m]_\mu + \frac{h}{24}\left\{55\left[\vec{V}^3\right]_\mu - 59\left[\vec{V}^2\right]_\mu + 37\left[\vec{V}^1\right]_\mu - 9\left[\vec{V}^0\right]_\mu\right\};$$

в) для выполнения шага коррекции необходимо на каждом ПЭ иметь $\left[A\tilde{\vec{y}}_{m+1} + \vec{f}_{m+1}\right]_\mu$, в вычислении которых принимают участие все ПЭ;

г) далее на каждом ПЭ выполняется шаг коррекции

$$[\vec{y}_{m+1}]_\mu = [\vec{y}_m]_\mu + \frac{h}{24}\left\{9([A]_\mu \tilde{\vec{y}}_{m+1} + \left[\vec{f}_{m+1}\right]_\mu) + 19\left[\vec{V}^3\right]_\mu - 5\left[\vec{V}^2\right]_\mu + \left[\vec{V}^1\right]_\mu\right\};$$

д) в заключение 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 возвращаемся на п. «г»,

если $\left\|\vec{y}_{m+1} - \tilde{\vec{y}}_{m+1}\right\| < \varepsilon$, то $\left[\vec{V}^0\right]_\mu = \left[\vec{V}^1\right]_\mu$, $\left[\vec{V}^1\right]_\mu = \left[\vec{V}^2\right]_\mu$,

$\left[\vec{V}^2\right]_\mu = \left[\vec{V}^3\right]_\mu$, $\left[\vec{V}^3\right]_\mu = [A]_\mu \cdot \vec{y}_{m+1} + \left[\vec{f}_{m+1}\right]_\mu$, присваиваем

$m = m + 1$ и возвращаемся на п. «б».

Произведем теоретическую оценку ускорения полученного параллельного алгоритма. Оценку будем производить по соотношению временных затрат на выполнение одного шага коррекции. Для последовательной версии получим $T_1 \approx 2n^2 t_a$. Для параллельной

версии имеем $T_p \approx 2\frac{n^2}{p}t_a + (p-1)\frac{n}{p}pt_{comm}$. Здесь t_a – время выполнения одной арифметической операции; t_{comm} – время на пересылку одного числа. Тогда ускорение можно оценить как

$$S_p = \frac{T_1}{T_p} \approx \frac{2n^2 t_a}{2\frac{n^2}{p}t_a + (p-1)nt_{comm}} = \frac{p}{1+\frac{(p-1)p\omega}{2n}},$$ где $\omega = t_{comm}/t_a$.

Выполним оценку ускорения.

Пусть $n = 100, \omega = 100$, $p = 10$. Тогда $S_{10} \approx \frac{10}{1+\frac{9\cdot 10\cdot 100}{2\cdot 100}} < 1$.

При $n = 1000, \omega = 100$, $p = 10$. Тогда $S_{10} \approx \frac{10}{1+\frac{9\cdot 10\cdot 100}{2\cdot 1000}} < 2$.

Таким образом, рассмотренная параллельная процедура решения системы ОДУ при небольшом числе используемых процессорных элементов имеет преимущество лишь для задач большой размерности.

Приведем текст параллельной программы решения систем ОДУ вида (7.1) с начальными условиями (7.2) с использованием многошаговых методов Адамса и схемы «предиктор–корректор».

```
      program predictor_corrector
c
      implicit none
      include 'mpif.h'
      include 'dim.h'
      integer i, rank, size, m, ierr, k
      double precision h, x, xk, time
      double precision y(n), y1(n), yy(0:3,n), ff(0:3,n), f1(n), y0(n)
      common/parallel/ rank, size, m
c Инициализация параллельной части программы
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank,
     $ ierr)
c Задание начального и конечного значений независимой
```

```
c переменной и шага
      x=0.0
      xk=10000.0
      h=0.001
c Декомпозиция уравнений системы
      m=n/size
      if(rank.lt.n-m*size) m=m+1
c
c Задание начальных условий
      do i=1,n
        y(i)=1.0
      end do
c
c Решение системы методом Рунге–Кутты для обеспечения
c корректного старта многошаговых методов: yy – решения
c системы ОДУ при x0, x1, x2, x3
      call rk4(x, h, y, yy)
c Расчет правых частей системы ОДУ при x0, x1, x2, x3
      do k=0,3
       do i=1,n
        y1(i)=yy(k,i)
       end do
       call rp(x-(3-k)*h, y1, f1)
       do i=1,m
        ff(k,i)=f1(i)
       end do
      end do
c Распределение решения по активным процессам
      call mpi_scatter(y, m, mpi_double_precision,
     $                       y0, m, mpi_double_precision, 0,
     $                         mpi_comm_world, ierr)
c Засечка времени начала выполнения основного расчетного
c блока
      if(rank.eq.0) time=mpi_wtime()
c Главный цикл выполнения схемы "предиктор–корректор"
       call adams(x, h, y, y0, ff, xk)  !Adams method
c Засечка времени для определения продолжительности
c  работы
c параллельной  программы
      if(rank.eq.0) time=mpi_wtime()-time
```

```
      if(rank.eq.0) write(6,*) 'size=',size,' timeC=',time
      do i=1,m
       write(6,*) i,y(i),rank
      end do
      call mpi_finalize(ierr)
      end
c
      subroutine adams(x, h, y, y0, ff, xk)
!   Решение системы ОДУ методом Адамса по схеме
!   "предиктор–корректор"
!   n – количество уравнений
!   x – начальная точка (xm)
!   h – шаг
!   y – массив размерности n - решение при x+h (xm+1)
!   y0 – массив размерности n - решение при x
!   ff – массив значений правых частей ОДУ при
!   xm-3,xm-2,xm-1,xm
!   rp – имя п/п для расчета правых частей
      implicit none
      include 'mpif.h'
      include 'dim.h'
      integer i, k, m, rank, size, ierr, iter
      double precision x, h, y(n), y1(n), y2(n), f1(n), yy(0:3,n),
     $ ff(0:3,n), res_global, h4, res, y0(n), xk
      external rp
      common/parallel/ rank, size, m
c
      h4 = h/24
      iter = 0
      do while (x.lt.xk)
c  Шаг прогноза по методу Адамса–Башфорда четвертого
c  порядка
      do i=1,m
       y1(i)=y0(i)+h4*(55.0*ff(3,i)-59.0*ff(2,i)+
     $              37.0*ff(1,i)-9.0*ff(0,i))   !!!предиктор
      end do
c
      x=x+h
c  Шаг коррекции по методу Адамса–Моултона четвертого
c  порядка
```

```
      res_global=1d0
      do while (res_global.gt.1d-10)
c Сборка на каждом процессе вектора решения
       call rp(x, y1, f1)  !расчет значения правой части
       res=0d0
       do i=1,m
        y(i)=y0(i)+h4*(9.0*f1(i)+19.0*ff(3,i)-5.0*ff(2,i)+ff(1,i))
!!!коррекция
        res=dmax1(dabs(y(i)-y1(i))/dmax1(dabs(y(i)),1d-10),res)
!!!вычисление нормы
        y1(i)=y(i)
!!!подготовка к вычислению следующего приближения
       end do
c Вычисление нормы глобальной ошибки
       call mpi_allreduce(res, res_global, 1, mpi_double_precision,
     $                          mpi_max, mpi_comm_world, ierr)
       iter=iter+1
      end do
c  Подготовка для осуществления следующей итерации
c   метода
      do i=1,m
       ff(0,i)=ff(1,i)
       ff(1,i)=ff(2,i)
       ff(2,i)=ff(3,i)
       ff(3,i)=f1(i)
       y0(i)=y(i)
      end do
      end do
      return
      end
c
      subroutine rp(x, y, f)
c Задание правых частей ОДУ, параллельная версия
      implicit none
c
      include 'mpif.h'
      include 'dim.h'
      integer i, m, rank, size, ierr
      double precision x, y(n), f(n), s, s_global
      common/parallel/ rank, size, m
```

```
c
      s=0
      do i=1,m
        s=s-y(i)
      end do
      call mpi_allreduce(s, s_global, 1, mpi_double_precision,
     $                           mpi_sum, mpi_comm_world, ierr)
      do i=1,m
       f(i)=(-y(i)+x*cos(-x))/(i+rank*m)+s_global/n
      end do
      return
      end
c
      subroutine rp0(x, y, f)
c Задание правых частей ОДУ, последовательная версия
      implicit none
c
      include 'dim.h'
      integer i
      double precision x, y(n), f(n), s
C
      s=0
      do i=1,n
        s=s-y(i)
      end do
      do i=1,n
        f(i)=(-y(i)+x*cos(-x))/i+s/n
      end do
      return
      end
c
      subroutine rk4(x, h, y, yy)
c Решение системы ОДУ методом Рунге–Кутты четвертого
c  порядка
      implicit none
c
      include 'dim.h'
      integer i, k, rank, size, m
      double precision y(n), k1(n), k2(n), k3(n), k4(n), y1(n),
     $ yy(0:3,n), x, h
```

```
      common/parallel/ rank, size, m
      external rp0
! процедура вычисления правых частей уравнений системы
! ОДУ
c Занесение начальных условий в массив yy
      do i=1,n
       yy(0,i)=y(i)
      end do
c
      k=1
!k – счетчик номера шага метода для занесения решения в
!массив yy
      do while (x.lt.3*h)
       call rp0(x, y, k1)
       do i=1,n
        y1(i)=y(i)+k1(i)*h/2
       end do
       call rp0(x+h/2, y1, k2)
       do i=1,n
        y1(i)=y(i)+k2(i)*h/2
       end do
       call rp0(x+h/2, y1, k3)
       do i=1,n
        y1(i)=y(i)+k3(i)*h
       end do
       call rp0(x+h, y1, k4)
       do i=1,n
        y(i)=y(i)+h*(k1(i)+2*k2(i)+2*k3(i)+k4(i))/6
       end do
       x=x+h
       do i=1,n
        yy(k,i)=y(i)
       end do
       k=k+1
      end do
      return
      end
```

В программе решается задача

$\frac{dy_i}{dt} = (-y_i + t\cos(-t))/i - \frac{1}{n}\sum_{j=1}^{n} y_j, i = 1,...,n$ с начальными условиями $y_i(0) = 1, i = 1,...,n$.

При ее запусках на разном числе используемых процессов при n=1000 были получены следующие результаты таймирования выполнения программы (в с):

Size = 1 time = 326,
Size = 2 time = 182,
Size = 4 time = 118,
Size = 5 time = 153,
Size = 10 time = 177.

Эти данные указывают на большие коммуникационные затраты созданного параллельного алгоритма при рассмотренном объеме вычислительной работы. Ускорение параллельной версии получается в случае, когда вычислительные затраты на расчет правых частей превосходят время, необходимое для сборки на всех процессах вектора прогноза решения $\tilde{\tilde{y}}_{m+1}$.

7.6 Задания

1. Используя представленную в п. 7.5 программу, решить систему линейных однородных ОДУ вида (7.1), в которой матрица является нижнетреугольной с единичными ненулевыми элементами. В качестве начальных условий принять $y_i(0) = 1, i = 1, n$. Для обеспечения равномерной загрузки процессоров применить циклическую схему распределения уравнений по процессам. На основе тестовых расчетов при заданном n определить значение параметра ω, при котором теоретические оценки ускорения наилучшим образом соответствуют расчетным данным.

2. Используя представленную в п. 7.5 программу, решить систему линейных однородных ОДУ вида (7.1), в которой матрица является верхнетреугольной с единичными ненулевыми элементами. В качестве начальных условий принять $y_i(0) = 1, i = 1, n$. Для обеспечения равномерной загрузки процессоров применить схему с отражениями для распределения уравнений по процессам. На основе тестовых расчетов при заданном n определить значение параметра

ω, при котором теоретические оценки ускорения наилучшим образом соответствуют расчетным данным.

3. Используя представленную в п. 7.5 программу, решить систему линейных неоднородных ОДУ вида (7.1), в которой матрица является трехдиагональной $(a_{ii} = -4, a_{ii\pm1} = 1)$, $f_i(t) = \int_0^t e^{-(x-i)^2} dx$. В качестве начальных условий принять $y_i(0) = 1, i = 1, n$. Для вычисления интеграла использовать формулу трапеций. Исследовать ускорение и эффективность параллельной программы в зависимости от размерности задачи и числа используемых процессов.

4. Используя представленную в п. 7.5 программу, решить систему линейных неоднородных ОДУ вида (7.1), в которой симметричная матрица имеет вид

$$\begin{pmatrix} n & n-1 & \dots & 1 \\ n-1 & n & \dots & 2 \\ \vdots & \vdots & \ddots & n-1 \\ 1 & 2 & \dots & n \end{pmatrix}, \text{ а } f_i(t) = \sum_{k=1}^{\infty} \frac{\cos(k)}{(k+i)!}.$$

В качестве начальных условий принять $y_i(0) = 1, i = 1, n$. Сумму вычислять с точностью $\varepsilon = 10^{-5}$. Исследовать ускорение и эффективность параллельной программы в зависимости от размерности задачи и числа используемых процессов.

5. Используя представленную в п. 7.5 программу, решить обыкновенное дифференциальное уравнение n-го порядка: $y^{(n)} = \cos(t)$. В качестве начальных условий принять $y^{(i)}(0) = 1, i = 0, n-1$. Исследовать ускорение и эффективность параллельной программы в зависимости от размерности задачи и числа используемых процессов.

6. Используя представленную в п. 7.5 программу, решить систему линейных однородных ОДУ вида (7.1), в которой матрица является ленточной с шириной верхней ленты $n/2$ и нижней $n/3$ и с единичными ненулевыми элементами. В качестве начальных условий принять $y_i(0) = 1, i = 1, n$. Для обеспечения равномерной загрузки процессоров применить схему с отражениями для распределения уравнений по процессам. На основе тестовых расчетов при заданном n определить значение параметра ω, при котором теоретиче-

ские оценки ускорения наилучшим образом соответствуют расчетным данным.

7. Используя представленную в п. 7.5 программу, решить обыкновенное дифференциальное уравнение n-го порядка: $y^{(n)} + ty^{(n-1)} + ... + t^{n-1}y' + t^n y = 1$. В качестве начальных условий принять $y^{(i)}(0) = 1, i = 0, n-1$. Исследовать ускорение и эффективность параллельной программы в зависимости от размерности задачи и числа используемых процессов.

8. Решить систему линейных неоднородных ОДУ вида (7.1), в которой матрица является пятидиагональной ($a_{ii} = -4, a_{ii\pm1} = 1, a_{ii\pm2} = 1$), а $f_i(t) = \int_0^t \cos(ix)dx$. В качестве начальных условий принять $y_i(0) = 1, i = 1, n$. Для вычисления интеграла использовать формулу Симпсона. Исследовать ускорение и эффективность параллельной программы в зависимости от размерности задачи и числа используемых процессов.

9. Эволюция системы N гравитирующих тел (материальных точек) описывается следующей системой уравнений:

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i; \frac{d\vec{v}_i}{dt} = \sum_{\substack{j=1 \\ j\neq i}}^{N} Gm_j \frac{(\vec{r}_j - \vec{r}_i)}{\left|\vec{r}_j - \vec{r}_i\right|^3},\ i = 1, ..., N,$$

где $m_i, \vec{r}_i, \vec{v}_i$ – масса, радиус-вектор и скорость i-го тела соответственно (i изменяется от 1 до N), G=6,6742·10^{-11} – гравитационная постоянная. Массы тел, а также их положения и скорости в начальный момент времени считаются известными. (Для больших N используйте датчик случайных чисел). Необходимо найти положения и скорости всех частиц в произвольный момент времени. Исследовать ускорение и эффективность параллельной программы в зависимости от размерности задачи и числа используемых процессов.

10. Составить параллельную программу для решения системы линейных однородных ОДУ вида (7.1) методом Рунге–Кутты четвертого порядка, описанным в п. 7.3. Рассмотреть случай диагональной матрицы ($a_{ii} = -1/i$). В качестве начальных условий принять $y_i(0) = 1, i = 1, n$. Исследовать ускорение и эффективность параллельной программы в зависимости от размерности задачи и числа используемых процессов.

8 ПАКЕТ PETSc ДЛЯ РЕШЕНИЯ УРАВНЕНИЙ В ЧАСТНЫХ ПРОИЗВОДНЫХ

Библиотека PETSc (Portable, Extensible Toolkit for Scientific Computation, http://www-unix.mcs.anl.gov/petsc/) является мощным средством для численного решения (на компьютерах как с параллельной, так и последовательной архитектурой) дифференциальных уравнений в частных производных (ДУЧП) и связанных с этим проблем.

PETSc может использоваться в программах на языках C, C++ и Fortran. Для обеспечения межпроцессорных обменов в пакете используется стандарт MPI (Message Passing Interface, http://www.mpi-forum.org/).

PETSc – это пакет с открытым исходным кодом, доступным для свободного скачивания (http://www.mcs.anl.gov/petsc). История PETSc началась в сентябре 1991 г. По сегодняшний момент с официального сайта он был загружен свыше 8 500 раз. PETSc перенесён на параллельные платформы, поддерживающие MPI, начиная с массивно-параллельных систем (Cray T3E, IBM SP, HP 9000, Sun Enterprise, SGI Origin) и заканчивая сетями рабочих станций (Compaq, HP, IBM, SGI, Sun). В создании пакета PETSc принимали участие разработчики MPICH.

PETSc состоит из набора компонент (модулей), каждая из которых работает с некоторым семейством объектов (например, с векторами или матрицами) и операциями, которые можно выполнять над этими объектами.

PETSc-модули используют следующие методы и объекты:

- множества индексов для индексации векторов, перенумерации и т.д.;
- векторы;
- матрицы (в том числе и разреженные);
- распределенные массивы;
- методы подпространств Крылова;
- средства решения нелинейных уравнений (основанные на использовании разнообразных методов Ньютона).

PETSc имеет различные свойства, облегчающие его использование прикладными программистами. Пользователи могут создавать параллельные программы для решения ДУЧП без написания большого количества явных процедур передачи сообщений самостоятельно. Параллельные векторы и разреженные матрицы могут быть

легко и эффективно собраны с использованием механизмов, реализованных в пакете. Более того, PETSc открывает большие возможности для контроля хода решения задачи во время исполнения программы без какого-либо дополнительного пользовательского программного кода. Опции пакета включают в себя управление выбором солвера (решателя), предобусловливателя, параметров задачи.

PETSc использует стандарт MPI для межпроцессорных обменов, что позволяет включать в PETSc-программы процедуры MPI. Однако пользователь изолирован от деталей передачи сообщений внутри PETSc, поскольку они не видимы внутри параллельных объектов (векторов, матриц).

Ниже представлено краткое описание порядка использования библиотеки PETSc в FORTRAN-программах.

8.1 Компиляция и запуск PETSc-программ

Чтобы написать новую программу с использованием PETSc, рекомендуется предварительно ознакомиться с одним из примеров соответствующего класса задач. Примеры PETSc-программ расположены в *${PETSC_DIR}/src/<library>/*, где <library> означает одну из библиотек PETSc, например snes или ksp.

Для компилирования PETSc-программы на кластере ТГУ СКИФ Cyberia можно воспользоваться скриптом petsc:

petsc petsc_program_name.F ,

где *petsc_program_name.F* – название программы на языке Fortran.

Использование расширения *.F* вместо стандартного *.f* является обязательным.

После выполнения скрипта в текущей директории *($pwd*) будет создана исполняемая программа с именем *petsc_program_name.*

Все PETSc-программы базируются на стандарте MPI. Поэтому чтобы выполнить параллельную программу PETSc, достаточно запустить:

mpirun -np p ./petsc_program_name,

где *p* – число процессоров.

Эта команда запустит программу *petsc_program_name* на *p* процессорах. Откомпилированная PETSc-программа также может быть запущена с использованием некоторых опций, которые можно посмотреть, запустив программу так:

./petsc_program_name -help

8.2 Структура PETSc-программ

Все PETSc-программы должны включать заголовочные файлы, необходимые для используемых компонентов PETSc, следующим образом:

#include "include/finclude/includefile.h",

где *includefile.h* может быть следующим:

petsc.h – содержит основные процедуры PETSc;

petscvec.h – содержит процедуры для работы с векторами;

petscmat.h – содержит процедуры для работы с матрицами;

petscpc.h – доступные предобуславливатели;

petscksp.h – методы подпространств Крылова;

petscsys.h – содержит системные процедуры PETSc (процедура для работы со случайными числами);

petscviewer.h – содержит параметры и процедуры для просмотра данных.

Необходимо явно перечислить каждый из включенных заголовочных файлов. Оператор #include должен начинаться с первой позиции строки в тексте программы. Этот подход реализуется с использованием CPP препроцессора. (Знакомство с CPP препроцессором не нужно для написания PETSc-Fortran-программ.)

PETSc-программа начинается с обращения к процедуре PETSCINITIALIZE для инициализации себя и MPI. Процедура PETSCINITIALIZE автоматически вызывает процедуру MPI_INIT, если она не была прежде инициализирована. В определенных обстоятельствах, когда нужно инициализировать MPI напрямую, сначала вызывается MPI_INIT и затем выполняется обращение к PETSCINITIALIZE. По умолчанию процедура PETSCINITIALIZE добавляет к MPI_COMM_WORLD коммуникатор с фиксированным именем PETSC_COMM_WORLD. В большинстве случаев пользователю достаточно коммуникатора PETSC_COMM_WORLD, чтобы указать все процессы в данном выполнении, а коммуникатор PETSC_COMM_SELF указывает на одиночный процесс.

Все PETSc-программы должны вызывать процедуру PETSCFINALIZE как последний вызов из PETSc-библиотеки.

PETSCFINALIZE в свою очередь вызывает MPI_FINALIZE, если MPI был запущен вызовом PETSCINITIALIZE.

Рассмотрим простейший пример параллельной MPI-программы «Hello world!» с применением PETSc:

```
    program helloworld
    implicit none
    integer ierr, rank
#include "include/finclude/petsc.h"
    call PETSCINITIALIZE(PETSC_NULL_CHARACTER, ierr )
    call MPI_COMM_RANK(PETSC_COMM_WORLD, rank,ierr )
    print *, "Hello World!", "process=", rank
    call PETSCFINALIZE(ierr)
    end
```

Здесь PETSC_NULL_CHARACTER – используется для использования файла настроек по умолчанию; ierr – переменная, в которую возвращается код ошибки, если ierr=0, то оператор проработал успешно; rank – переменная, содержащая номер активного процесса.

8.3 Векторы

8.3.1 Работа с векторами

PETSc содержит два базисных векторных типа: последовательный и параллельный (для MPI). Для описания вектора в PETSc используется тип Vec, например:

```
VEC x
```

Базовым типом является double precision. Чтобы создать последовательный вектор x с m компонентами, нужно использовать процедуру

```
SUBROUTINE VecCreateSEQ(PETSC_COMM_SELF,m,x,ierr)
INTEGER m, ierr
VEC x
```

Чтобы создать вектор x для параллельных вычислений, нужно задать число компонентов, которые будут храниться на каждом процессорном элементе, либо передать эту проблему PETSc. Для этого используется процедура

```
SUBROUTINE VecCreateMPI(comm, m, n, x, ierr)
INTEGER m, n, comm, ierr
VEC x
```

которая создает вектор, распределенный по всем процессам в коммуникаторе *comm*. Здесь *m* указывает число компонентов, которые нужно хранить в памяти на локальном процессе, а *n* есть общее число компонентов вектора. Вектор *x* можно распределить и автоматически, используя вместо *m* или *n* директиву PETSC_DECIDE. При автоматическом выборе с помощью PETSC_DECIDE можно указать только локальную или глобальную размерность вектора, но не обе вместе. Например:

```
Call VecCreateMPI(comm, PETSC_DECIDE, n, x, ierr)
```

автоматически распределит вектор *x* из *n* компонент по процессам.

Чтобы создать новый вектор *v* того же формата, что и существующий *x*, используется процедура

```
SUBROUTINE VecDuplicate(x, v, ierr)
Vec x, v
```

Когда вектор *x* больше не будет использоваться в программе, он удаляется процедурой

```
SUBROUTINE VecDestroy(x, ierr)
Vec x
```

Единственное значение *value* всем компонентам вектора *x* можно присвоить с помощью процедуры

```
SUBROUTINE VecSet(x, value, ierr)
INTEGER ierr
Vec x
Double precision value
```

Присвоение различных значений индивидуальным компонентам вектора более сложно, если нужно получить эффективный парал-

лельный код. Присвоение ряда компонентов есть двухшаговый процесс: сначала вызывают процедуру

```
SUBROUTINE VecSetValues(x, n, indices, values, INSERT_VALUES, ierr)
Vec x
INTEGER n, indices(k1), ierr
Double precision values(k2)
```

любое число раз на один или все используемые процессы. Аргумент *n* – количество компонент, установленных в этом присваивании. Целочисленный массив *indices* содержит индексы глобальных компонентов, а *values* – массив добавляемых значений. Часто необходимо не вставить элементы в вектор, а сложить значения, для этого вместо INSERT_VALUES используется ADD_VALUES. Например:

```
Call VecSetValues(x, n, indices, values, ADD_VALUES, ierr)
```

Любой параллельный процесс может определить любой компонент вектора, при этом PETSc гарантирует, что присваиваемые значения сохранятся в правильной ячейке. После того как значения размещены с помощью VecSetValues, нужно вызвать две процедуры

```
SUBROUTINE VecAssemblyBegin(x,ierr)
SUBROUTINE VecAssemblyEnd(x,ierr)
Vec x
INTEGER ierr
```

чтобы выполнить необходимую межпроцессорную передачу нелокальных компонент.

Для совмещения обменов и вычислений код пользователя может выполнять любую последовательность других действий между этими двумя вызовами, пока сообщения находятся в состоянии передачи.

Вектор можно вывести на экран с помощью процедуры

```
SUBROUTINE VecView(x,
PETSC_VIEWER_STDOUT_WORLD)
Vec x
```

8.3.2 Основные векторные операции

Таблица 8.1 Представление формальных параметров векторных операций

Название процедуры	Операция
VecAXPY(y, a, x, ierr)	$y = y + a * x$
VecAYPX(y, a, x, ierr)	$y = x + a * y$
VecWAXPY(w, a, x, y, ierr)	$w = a * x + y$
VecAXPBY(y, a, b, x, ierr)	$w = a * x + b * y$
VecScale(x, a, ierr)	$x = a * x$
VecNorm(x, normtype, r, ierr) *normtype* принимает одно из следующих значений: NORM_1 ($\sum_i \lvert x_i \rvert$), NORM_2 ($\sqrt{\sum_i x_i^2}$) или NORM_INFINITY ($\max \lvert x_i \rvert$)	$r = \lVert x \rVert_{type}$
VecSum(x, r, ierr)	$r = \sum_i x_i$
VecCopy(x, y, ierr)	$x=y$
VecSwap(x, y, ierr)	$y = x \; while \; x.eq.y$
VecPointwiseMult(x, y, w, ierr)	$w_i = x_i * y_i$
VecPointwiseDivide(x, y, w, ierr)	$w_i = x_i / y_i$
VecMax(x, PETSC_NULL, r, ierr)	$r = \max (x_i)$
VecMin(x, PETSC_NULL, r, ierr)	$r = \min (x_i)$
VecAbs(x, ierr)	$x_i = \lvert x_i \rvert$
VecReciprocal(x, ierr)	$x_i = \frac{1}{x_i}$
VecShift(s, x, ierr)	$x_i = x_i + s$

В табл. 8.1 используются следующие описания типов формальных переменных:

Vec x, y, w
DOUBLE PRECISION a, b, r, s
INTEGER ierr

Для параллельных программ распределение вектора x по процессам можно осуществить с помощью вызова процедуры

Call VecGetOwnershipRange(x, istart, iend, ierr)

Аргумент istart указывает на номер первой компоненты вектора, принадлежащий локальному процессу, а аргумент iend показывает на единицу больший номер, чем номер последней размещенной компоненты, принадлежащей локальному процессу. Эта команда полезна, например, при сборке параллельных векторов.

8.3.3 Пример

Рассмотрим пример с использованием параллельной программы, использующей библиотеку PETSc. Вычислим $\vec{y} = \vec{y} + \|\vec{x}\|_{NORM_2} * \vec{x}$, $y_i = 3$, $x_i = i$, $i = 1..n$

```
      program main
      implicit none
#include "include/finclude/petsc.h"
#include "include/finclude/petscis.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscviewer.h"
      Integer ierr, n, i, istart, iend, rank
      double precision v, r
      Vec x, y
c     устанавливаем размер вектора
      n=10
c     инициализируем PETSc
      Call PetscInitialize(PETSC_NULL_CHARACTER, ierr)
      Call MPI_Comm_rank(PETSC_COMM_WORLD, rank, ierr)
      Call VecCreate(PETSC_COMM_WORLD, x, ierr)
      Call VecSetSizes(x,PETSC_DECIDE, n, ierr)
      Call VecSetType(x,VECMPI, ierr)
      Call VecDuplicate(x, y, ierr)
      Call VecGetOwnershipRange(x, istart, iend, ierr)
      do i=istart,iend-1
       v=dble(i)
       Call VecSetValues(x, 1, i, v, INSERT_VALUES, ierr)
      end do
```

```
 Call VecAssemblyBegin(x, ierr)
 Call VecAssemblyEnd(x, ierr)
 v=3.0
 Call VecSet(y, v, ierr)
 Call VecNorm(x, NORM_2, r, ierr)
 if (rank .eq. 0) write (*,*) 'r=',r
 Call VecAXPY(y, r, x, ierr)
 if (rank .eq. 0) write (*,*) 'Vector X'
 Call VecView(x, PETSC_VIEWER_STDOUT_WORLD, ierr)
 if (rank .eq. 0) write (*,*) 'Vector Y'
 Call VecView(y, PETSC_VIEWER_STDOUT_WORLD, ierr)
 Call VecDestroy(x, ierr)
 Call VecDestroy(y, ierr)
 Call PetscFinalize(ierr)
 end
```

После запуска программы на 4 процессорах

mpirun -np 4 ./ex1.exe

получим

```
r=  16.881943
 Vector X
Process [0]
0
1
2
Process [1]
3
4
5
Process [2]
6
7
Process [3]
8
9
 Vector Y
Process [0]
3
19.8819
```

```
36.7639
Process [1]
53.6458
70.5278
87.4097
Process [2]
104.292
121.174
Process [3]
138.056
154.937
```

8.4 Матрицы

8.4.1 Работа с матрицами

Использование матриц в PETSc осуществляется подобно векторам. По умолчанию матрицы в PETSc представлены в общем AIJ формате. Матрицы в PETSc имеют тип Mat, с базовым типом DOUBLE PRECISION, например:

```
Mat A
```

Пользователь может создавать новую матрицу *A* для последовательных вычислений, которая имеет *m* строк и *n* столбцов, с помощью процедуры:

```
SUBROUTINE MatCreateSeqAIJ(PETSC_COMM_SELF, m, n,
                           nz, nnz, A, ierr)
Mat A
INTEGER m, n, nz, nnz, ierr
```

Здесь *nz* – количество ненулевых элементов в строке, *nnz* – в общем случае массив, содержащий число ненулевых элементов в различных строках. Для динамического распределения памяти можно установить *nz*=0 и *nnz*=PETSC_NULL.

Для создания матрицы в параллельной программе с числом строк *m* и числом столбцов *n* следует использовать процедуру

```
SUBROUTINE MatCreateMPIAIJ (comm, m1, n1, m, n, d_nz,
  $                         d_nnz, o_nz, o_nnz, A, ierr)
Mat A
INTEGER comm, m1, n1, m, n, d_nz, d_nnz, o_nz, o_nnz, ierr
```

Матрица будет распределена по всем процессам в коммуникаторе *comm*, где *m1* – указывает число локальных строк (может быть вычислено автоматически, если указано число *m* с помощью директивы PETSC_DECIDE); *n1* – это значение, совпадающее с локальным размером, используемым при создании вектора *x*. Оно может быть автоматически вычислено с помощью PETSC_DECIDE, если задано число *n*. *d_nz* – число ненулевых элементов в строке диагональной части локальной подматрицы; *d_nnz* – массив, содержащий количество ненулевых элементов в строках диагональной части локальной подматрицы; *o_nz* – число ненулевых элементов в строке недиагональной части локальной подматрицы; *o_nnz* – массив, содержащий количество ненулевых элементов в строках недиагональной части локальной подматрицы. Пользователю необходимо установить аргументы *d_nz*=0, *o_nz*=0, *d_nnz*=PETSC_NULL и *o_nnz*=PETSC_NULL для PETSc, чтобы динамически разместить элементы в памяти.

Номер в коммуникаторе MPI определяет абсолютное упорядочивание блоков. Это означает, что процесс с номером 0 в коммуникаторе, определенном для MatCreateMPIAIJ, содержит верхний блок строк матрицы; *i*-й процесс в коммуникаторе содержит *i*-й блок строк матрицы.

Наиболее просто задать матрицу можно с использованием командной строки:

```
Call MatCreate(comm, A, ierr)
Call MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, m, n,
$                ierr)
Call MatSetFromOptions(A, ierr)
```

Соответственно программа в таком случае должна быть запущена с указанием типа матрицы опцией *-mat_type seqaij* (последовательная программа) или *-mat_type mpiaij* (параллельная программа). Или достаточно просто указать тип матрицы вызовом процедуры

Call MatSetType(A, MATMPIAIJ, ierr) – для параллельной матрицы и

Call MatSetType(A, MATMPISEQ, ierr) – для последовательной.

Значения матрицы затем могут быть установлены процедурой

```
SUBROUTINE MatSetValues(A, m, im, n, in, values,
                                  INSERT_VALUES, ierr)
```

или

```
SUBROUTINE MatSetValues(A, m, im, n, in, values,
$                                ADD_VALUES, ierr)
```

Здесь

m – число строк, *im* – их глобальные индексы;

n – число столбцов, *in* – их глобальные индексы.

В процедуре MatSetValues используется соглашение стандарта языка С, в котором строки и столбцы нумеруются от нуля. Массив values является логически двумерным и содержит вставляемые значения. По умолчанию эти значения расположены по строкам.

Хотя можно вставлять значения в матрицу вне зависимости от того, какой процессор в настоящий момент содержит их, все же рекомендуется генерировать большую часть элементов на том процессоре, где эти данные будут храниться. Для этого используется процедура

```
SUBROUTINE MatGetOwnershipRange(A, istart, iend, ierr)
Mat A
INTEGER istart, iend, ierr
```

Аргумент *istart* указывает на первую строку, принадлежащую локальному процессу, а аргумент *iend*-1 показывает на последнюю строку, принадлежащую локальному процессу.

После того как все элементы помещены в матрицу, она должна быть обработана парой процедур:

```
SUBROUTINE MatAssemblyBegin(A,
$                                MAT_FINAL_ASSEMBLY, ierr)
Mat A
INTEGER ierr
SUBROUTINEatAssemblyEnd(A,
$                                MAT_FINAL_ASSEMBLY,ierr)
Mat A
INTEGER ierr
```

Если матрица имеет разреженную структуру, то после выполнения процедур сборки матрицы сжимаются и могут быть использованы для матрично-векторного умножения или других матричных операций. Ввод новых значений в матрицу после этого может потребовать перераспределения памяти и дополнительных межпроцессорных пересылок. Поэтому не следует устанавливать элементы матрицы после обращения к конечным процедурам сборки.

Таблица 8.2 Основные матричные операции

Название процедуры	Операция
MatAXPY(a, x, y, DIFFERENT_NONZERO_PATTERN,ierr) Mat X,Y Double precision a INTEGER ierr	$Y = Y + a * X$
MatMult(A, x, y, ierr) Mat A Vec x,y INTEGER ierr	$y = A * x$
MatMultAdd(A, x, y, z, ierr) Mat A Vec x, y, z INTEGER ierr	$z = y + A * x$
MatMultTrans(A, x, y, ierr) Mat A Vec x, y INTEGER ierr	$y = A^T * x$
MatMultTransAdd(A, x, y, z, ierr) Mat A Vec x, y, z INTEGER ierr	$z = y + A^T * x$
MatScale(A, l, ierr) Mat A Double precision l INTEGER ierr	$A = l * A$
MatCopy(A, B, str, ierr) Mat A, B INTEGER ierr *str* принимает одно из следующих значений: SAME_NONZERO_PATTERN(одинаковый “не нулевой” шаблон) или DIFFERENT_NONZERO_PATTERN(различный “не нулевой” шаблон)	$B = A$
MatGetDiagonal(A, x, ierr) Mat A Vec x INTEGER ierr	$x = diag(A)$
MatTranspose(A, B, ierr) Mat A, B INTEGER ierr	$B = A^T$
MatShift(a, Y, ierr) Double precision a Mat Y INTEGER ierr	$Y = Y + a * I$, I – единичная матрица

В табл. 8.2 приведены некоторые операции для выполнения операций с участием матриц.

Вывести на экран матрицу можно, воспользовавшись процедурой

```
      SUBROUTINE MatView(A,
     $                   PETSC_VIEWER_STDOUT_WORLD, ierr)
      Mat A
      INTEGER ierr
```

По окончании использования матрицы в программе её необходимо удалить процедурой

```
      Call MatDestroy(A, ierr)
      Mat A
```

8.4.2 Пример

Умножение матриц размером 1000x1000. Значения матриц генерируются с помощью датчика случайных чисел.

```
      program main
      USE IFPORT
      implicit none
#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscksp.h"
#include "include/finclude/petscpc.h"

      Mat A, B, C
      INTEGER n
      PARAMETER (n=1000)
      INTEGER i, j, istart, iend, ierr, rank, m(0:n-1)
      double precision t, time1, time2
c
      Call PetscInitialize(PETSC_NULL_CHARACTER, ierr)
      Call MPI_COMM_RANK(PETSC_COMM_WORLD, Rank,
     $                   Ierr)
c     Создаём матрицу
      Call MatCreate(PETSC_COMM_WORLD, A, ierr)
      Call MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE,
     $                 n, n, ierr)
```

```
      Call MatSetFromOptions(A, ierr)
      Call MatCreate(PETSC_COMM_WORLD, B, ierr)
      Call MatSetSizes(B, PETSC_DECIDE, PETSC_DECIDE,
     $                 n, n, ierr)
      Call MatSetFromOptions(B, ierr)
      CALL RANDOM_SEED()
      Call MatGetOwnershipRange(A, Istart, Iend, ierr)
      do i=istart,iend-1
       do j=0,n-1
        CALL RANDOM_NUMBER(t)
        if (t.ne.0) Call MatSetValue(A, i, j, dble(t), INSERT_VALUES,
     $                               ierr)
       end do
      end do
      Call MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY, ierr)
      Call MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY, ierr)
      Call MatGetOwnershipRange(B, Istart, Iend, ierr)
      do i=istart,iend-1
       do j=0,n-1
        CALL RANDOM_NUMBER(t)
        if (t.ne.0) Call MatSetValue(B, i, j, dble(t), INSERT_VALUES,
     $                               ierr)
       end do
      end do
      Call MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY, ierr)
      Call MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY, ierr)
      write(6,*) istart, iend, rank
       time1=MPI_WTIME()
      Call MatMatMult(A, B, MAT_INITIAL_MATRIX, dble(1), C,
     $                ierr)
      time2=MPI_WTIME()
      time1=time2-time1
      write(*,*) time1
c     Обрабатываем матрицу
      Call MatView(A, PETSC_VIEWER_STDOUT_WORLD, ierr)
      Call MatView(B, PETSC_VIEWER_STDOUT_WORLD, ierr)
      Call MatView(C, PETSC_VIEWER_STDOUT_WORLD, ierr)
c     Уничтожаем все использовавшиееся PETSc объекты
      Call MatDestroy(A, ierr)
      Call MatDestroy(B, ierr)
```

```
      Call MatDestroy(C, ierr)
c     Завершаем работу PETSc
      Call PetscFinalize(ierr)
      end
```

График зависимости полученного ускорения параллельной программы от числа используемых процессов представлен на рис. 8.1.

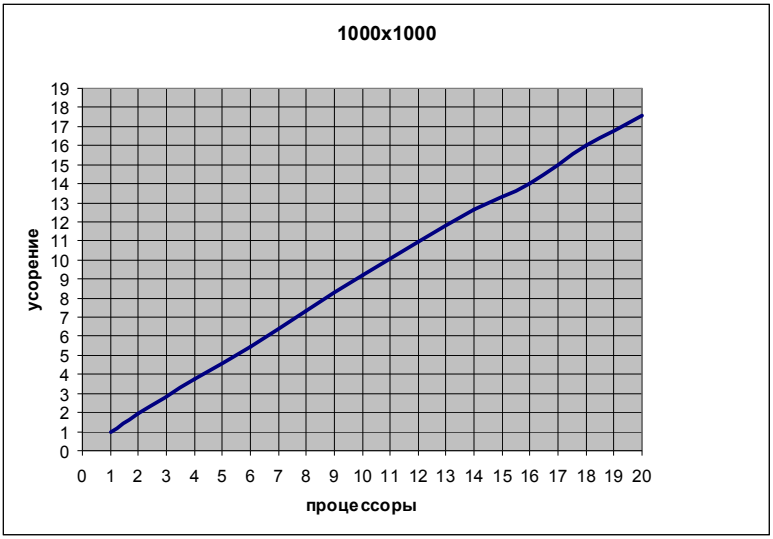

Рис. 8.1 Ускорение параллельной PETSc-программы умножения матриц 1000x1000

8.5 Методы решений систем линейных уравнений

Модуль KSP (Krylov subspace method – метод подпространств Крылова) является ядром PETSc, поскольку он обеспечивает унифицированный и эффективный доступ ко всем средствам для решения систем линейных алгебраических уравнений (СЛАУ). KSP предназначен для решения несингулярных систем вида

$$Ax = b\ ,$$

где *A* обозначает матричное представление линейного оператора, *b* – вектор правых частей, *x* – вектор решения. Во время решения могут быть выбраны конкретные способы решения и связанные с ними опции.

Комбинация метода подпространств Крылова (KSP) и предобуславливателя (PC – preconditioner) находится в центре самых современных программ для решения линейных систем итерационными методами.

Модуль KSP предоставляет многие популярные итерационные методы на основе подпространств Крылова: Conjugate Gradient, GMRES, CG-Squared, Bi-CG-Stab и др.

Модуль PC включает разнообразные предобуславливатели: Block Jacobi, Overlapping Additive Schwarz, ICC, ILU via BlockSolve95, ILU(k), LU (прямой метод, работает только на одном процессоре), Arbitrary matrix и др.

Использование модуля KSP задается процедурой:

```
SUBROUTINE KSPCreate(comm, ksp, ierr)
KSP ksp
INTEGER comm, ierr
```

Заметим, что решатель имеет тип KSP. Перед решением линейной системы с помощью KSP необходимо вызвать следующую процедуру, чтобы задать матрицы, связанные с линейной системой:

```
Call KSPSetOperators(ksp, A1, A2,
$                    DIFFERENT_NONZERO_PATTERN, ierr)
KSP ksp
MAT A1, A2
INTEGER ierr
```

*A*1 – матрица, определяющая линейную систему;

*A*2 – предобуславливающая матрица.

Установить метод решения можно либо из командной строки, добавив в программу вызов следующей процедуры:

```
SUBROUTINE KSPSetFromOptions(ksp, ierr)
KSP ksp
INTEGER ierr
```

и воспользовавшись опцией *-ksp_type*, допустимые значения: *richardson, chebychev, cg, gmres, bcgs* (полный список можно посмотреть, запустив программу с опцией *-help*) либо непосредственно в программе с помощью процедуры

```
Call KSPSetType(ksp, type, ierr)
```

KSP ksp
INTEGER ierr

где type соответствует типам, указываемым в *-ksp_type*.

Предобуславливатель устанавливается аналогично:

Call PCSetFromOptions(pc, ierr)
PC pc
INTEGER ierr

опция *-pc_type*, допустимые значения которой *none, jacobi, sor, lu, ilu* (для полного списка *-help*) или

Call PCSetType(pc, type, ierr)
PC pc
INTEGER ierr

Для решения линейной системы выбранным методом необходимо воспользоваться процедурой

SUBROUTINE KSPSolve(ksp, b, x, ierr)
KSP ksp
Vec x, b
INTEGER ierr

где *b* и *x* соответственно обозначают вектор *b* и вектор решения.

Для получения количества итераций нужно воспользоваться процедурой

Call KSPGetIterationNumber(ksp, its, ierr)
KSP ksp
INTEGER its, ierr

Параметр *its* содержит либо номер итерации, на которой критерий сходимости был успешно выполнен, или номер итерации, на которой была обнаружена расходимость или прерывание.

Если решатель (солвер) KSP больше не нужен, он удаляется командой

Call KSPDestroy(ksp, ierr)
KSP ksp
INTEGER ierr

8.6 Пример

Найти численно методом конечных разностей решение уравнения Лапласа

$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$, в области ($0 < x < 1$, $0 < y < 1$).

На границе квадрата выполняются условия Дирихле: $u_\Gamma = 1$.

В результате применения метода конечных разностей на сетке $\varpi_h = \{(ih, jh), i, j = 0, ..., n; h = 1/n\}$ получим следующую систему линейных алгебраических уравнений относительно неизвестных $\{u_{ij}\}$:

$$\begin{cases} u_{i+1j} + u_{i-1j} + u_{ij+1} + u_{ij-1} - 4u_{ij} = 0, i = 1, ..., n-1; j = 1, ..., n-1, \\ u_{ij} = 1, (i = 0) \cap (i = n) \cap (j = 0) \cap (j = n). \end{cases}$$

```
    program main
     implicit none
#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscksp.h"
#include "include/finclude/petscpc.h"

      Vec x, b, u
! приближенное решение, вектор правых частей, точное
! решение
      Mat A      ! матрица, определяющая линейную систему
      KSP ksp    ! линейный солвер
      PC pc      ! предобуславливатель
      Double precision norm ! норма ошибки
      INTEGER i, j, Ii, Jj, Istart, Iend, m, n, its, ierr, rank
      Double precision  v, one

      Call PetscInitialize(PETSC_NULL_CHARACTER, ierr)
      Call MPI_COMM_RANK(PETSC_COMM_WORLD, Rank,
     $                              Ierr)
c     Зададим размеры матрицы
      m = 8
      n = 7
c     Вычисляем матрицу и вектор правых частей,
c     которые определяют линейную систему Ax = b

c     Создаём матрицу, распределенную по активированным
c     процессам
      Call MatCreate(PETSC_COMM_WORLD, A, ierr)
```

```
      Call MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, m*n,
     $                 m*n, ierr)
      Call MatSetFromOptions(A, ierr)

c     Определяем номера строк, распределенных по локальным
c      процессам
      Call MatGetOwnershipRange(A, Istart, Iend, ierr)
c     Заполним матрицу

      do Ii=istart,iend-1
       v = -1.0
       i = Ii/n
       j = Ii - i*n
       if (i.gt.0) then
         JJ = Ii - n
         Call MatSetValues(A, 1, Ii, 1, JJ, v, INSERT_VALUES, ierr)
       end if
       if (i.lt.(m-1)) then
        JJ = Ii + n
        Call MatSetValues(A, 1, Ii, 1, JJ, v, INSERT_VALUES, ierr)
       end if
       if (j.gt.0) then
        JJ = Ii - 1
        Call MatSetValues(A, 1, Ii, 1, JJ, v, INSERT_VALUES, ierr)
       end if
       if (j.lt.(n-1)) then
        JJ = Ii + 1
        Call MatSetValues(A, 1, Ii, 1, JJ, v, INSERT_VALUES, ierr)
       end if
       v = 4.0
       Call MatSetValues(A, 1, Ii, 1, Ii, v, INSERT_VALUES, ierr)
      end do

c     Обрабатываем матрицу
      Call MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY, ierr)
      Call MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY, ierr)
      Call MatView(A, PETSC_VIEWER_STDOUT_WORLD, ierr)

c     Создаём параллельные векторы
      Call VecCreate(PETSC_COMM_WORLD, u, ierr)
```

```
      Call VecSetSizes(u, PETSC_DECIDE, m*n, ierr)
      Call VecSetFromOptions(u, ierr)
      Call VecDuplicate(u, b, ierr)
      Call VecDuplicate(b, x, ierr)

c     Устанавливаем точное решение (u=1.0), а затем вычисляем
c     вектор правых частей
      one=1.0
      Call VecSet(u, one, ierr)
      Call MatMult(A, u, b, ierr)

c     Создаем линейный солвер
      Call KSPCreate(PETSC_COMM_WORLD, ksp, ierr)

c     Устанавливаем операторы. Матрица, определяющая
c     линейную систему, будет также предобуславливающей
c     матрицей
      Call KSPSetOperators(ksp, A, A,
     $                     DIFFERENT_NONZERO_PATTERN, ierr)

c     Метод решения будем устанавливать в командной строке
c      -ksp_type <type>
      Call KSPSetFromOptions(ksp, ierr)

c     Решаем СЛАУ
      Call KSPSolve(ksp, b, x, ierr)

c     Проверка решения
      Call VecAXPY(x, -one, u, ierr)
      Call VecNorm(x, NORM_2, norm, ierr)
      Call KSPGetIterationNumber(ksp, its, ierr)
       if (rank .eq. 0) then
        if (norm .gt. 1.e-12) then
         write(6,100) norm, its
        else
         write(6,110) its
        endif
       endif
  100  format('Norm of error ',e10.4,',iterations ',i5)
  110  format('Norm of error < 1.e-12,iterations ',i5)
```

```
      Call VecView(u, PETSC_VIEWER_STDOUT_WORLD, ierr)
c     Уничтожаем все использовавшиеся PETSc объекты
      Call KSPDestroy(ksp, ierr)
      Call VecDestroy(u, ierr)
      Call VecDestroy(x, ierr)
      Call VecDestroy(b, ierr)
      Call MatDestroy(A, ierr)

c     Завершаем работу PETSc
      Call PetscFinalize(ierr)
      end
```

Запустим программу на 4 процессорах, используя разные методы:

```
mpirun -np 4 ./laplas -ksp_type gmres
Norm of error 0.3097E-04,iterations    10
mpirun -np 4 ./laplas -ksp_type bcgs
Norm of error 0.5074E-04,iterations     6
mpirun -np 4 ./laplas -ksp_type cg
Norm of error 0.3140E-04,iterations    10
```

8.7 Задания

Решить задачу Дирихле для уравнения Пуассона. Численное решение получить в области G с границей Г, которая задается соответствующим образом в каждом варианте.

1. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=6(x+y);\ (x,y)\in \overline{G}=\{(x,y),\ \ 0<x<1,\ \ 0<y<1\};$

$u(0,y)=y^3,\ 0\le y\le 1;\ u(x,0)=x^3,\ 0\le x\le 1;$

$u(1,y)=1+y^3,\ 0\le y\le 1;\ u(x,1)=1+x^3,\ 0\le x\le 1.$

2. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=2e^{x+y};\ (x,y)\in \overline{G}=\{(x,y),\ \ 0\le x\le 0.5,\ \ 0\le y\le 1\};$

$u(0,y)=y^3,\ 0\le y\le 1;\ u(x,0)=e^x,\ 0\le x\le 0.5;$

$u(0.5,y)=\sqrt{e}\,e^y,\ 0\le y\le 1;\ u(x,1)=e^{1+x},\ 0\le x\le 0.5.$

3. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=\left(2-x^2\right)\sin y$;

$(x,y)\in\bar{G}=\left\{(x,y),\ \ 0\le x\le 1,\ \ 0\le y\le\dfrac{\pi}{2}\right\}$;

$u(0,y)=0,\ 0\le y\le\dfrac{\pi}{2}$; $u(x,0)=0,\ 0\le x\le 1$;

$u(1,y)=\sin y,\ 0\le y\le\dfrac{\pi}{2}$; $u\left(x,\dfrac{\pi}{2}\right)=x^2,\ 0\le x\le 1$.

4. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=\left(2-y^2\right)\cos x$;

$(x,y)\in\bar{G}=\{(x,y),\ \ 0\le x\le\pi,\ \ 0\le y\le 1\}$;

$u(0,y)=y^2,\ 0\le y\le 1$; $u(x,0)=0,\ 0\le x\le\pi$;

$u(\pi,y)=-y^2,\ 0\le y\le 1$; $u(x,1)=\cos x,\ 0\le x\le\pi$.

5. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=x\left(6+x^2\right)e^y$;

$(x,y)\in\bar{G}=\{(x,y),\ \ 0\le x\le 1,\ \ 0\le y\le 1\}$;

$u(x,0)=x^3,\ 0\le x\le 1$; $u(0,y)=0,\ 0\le y\le 1$;

$u(x,1)=2.7183x^3,\ 0\le x\le 1$; $u(1,y)=e^y,\ 0\le y\le 1$.

6. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=\left(2+y^2\right)e^x$;

$(x,y)\in\bar{G}=\{(x,y),\ \ 0\le x\le 2,\ \ 0\le y\le 1\}$;

$u(x,0)=0,\ 0\le x\le 2$; $u(0,y)=y^2,\ 0\le y\le 1$;

$u(x,1)=e^x,\ 0\le x\le 2$; $u(2,y)=7.38906y^2,\ 0\le y\le 1$.

7. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=2\left(e^{2x}+1\right)$;

$(x,y)\in\bar{G}=\{(x,y),\ \ 0\le x\le 0.5,\ \ 0\le y\le 2\}$;

$u(x,0)=\dfrac{1}{2}e^{2x},\ 0\le x\le 0.5$; $u(0,y)=0.5+y^2,\ 0\le y\le 2$;

$u(x,2)=\frac{1}{2}\left(e^{2x}+8\right),\ 0\le x\le 0.5;\ u\left(0.5,y\right)=1.35914+y^2,$

$0\le y\le 2$.

8. $\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=\cos y-\sin x;$

$(x,y)\in\overline{G}=\left\{(x,y),\ \ 0\le x\le\frac{\pi}{2},\ \ 0\le y\le\pi\right\};$

$u(x,0)=\sin x,\ 0\le x\le\frac{\pi}{2};\ u(0,y)=-\cos y,\ 0\le y\le\pi;$

$u(x,\pi)=1+\sin x,\ 0\le x\le\frac{\pi}{2};\ u\left(\frac{\pi}{2},y\right)=1-\cos y,\ 0\le y\le\pi.$

9. $\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0;\ (x,y)\in\overline{G}=\left\{(x,y),\ \ 0\le x\le\pi,\ \ 0\le y\le 2\right\};$

$u(x,0)=\sin x,\ 0\le x\le\pi;\ u(0,y)=0,\ 0\le y\le 2;$

$u(x,2)=7.38906\sin x,\ 0\le x\le\pi;\ u\left(\pi,y\right)=0,\ 0\le y\le 2.$

10. $\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0;\ (x,y)\in\overline{G}=\left\{(x,y),\ \ 0\le x\le 1,\ \ 0\le y\le\pi\right\};$

$u(x,0)=e^x,\ 0\le x\le 1;\ u(0,y)=\cos y,\ 0\le y\le\pi;$

$u(x,\pi)=-e^x,\ 0\le x\le 1;\ u\left(1,y\right)=2.71828\cos y,\ 0\le y\le\pi.$

11. $\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=-\left(\cos x+\sin y\right);$

$(x,y)\in\overline{G}=\left\{(x,y),\ \ 0\le x\le\frac{\pi}{2},\ \ 0\le y\le\frac{\pi}{2}\right\};$

$u(x,0)=\cos x,\ 0\le x\le\frac{\pi}{2};\ u(0,y)=1+\sin y,\ 0\le y\le\frac{\pi}{2};$

$u(x,\frac{\pi}{2})=1+\cos x,\ 0\le x\le\frac{\pi}{2};\ u\left(\frac{\pi}{2},y\right)=\sin y,\ 0\le y\le\frac{\pi}{2}.$

12. $\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=2+e^y;\ (x,y)\in\overline{G}=\left\{(x,y),\ \ 0\le x\le 1,\ \ 0\le y\le 2\right\};$

$u(x,0)=1+x^2,\ 0\le x\le 1;\ u(0,y)=e^y,\ 0\le y\le 2;$

$u(x,2) = x^2 + 7.38906$, $0 \le x \le 1$; $u(1,y) = 1 + e^y$, $0 \le y \le 2$.

13. $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 2 - \cos y$; $(x,y) \in \overline{G} = \{(x,y),\ 1 \le x \le 2,\ 0 \le y \le 1\}$;

$u(x,0) = 1 + x^2$, $1 \le x \le 2$; $u(1,y) = 1 + \cos y$, $0 \le y \le 1$;

$u(x,1) = 0.5403 + x^2$, $1 \le x \le 2$; $u(2,y) = 4 + \cos y$, $0 \le y \le 1$.

14. $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 2 - \sin y$;

$(x,y) \in \overline{G} = \left\{(x,y),\ 2 \le x \le 3,\ 0 \le y \le \dfrac{\pi}{2}\right\}$;

$u(x,0) = x^2$, $2 \le x \le 3$; $u(2,y) = 4 + \sin y$, $0 \le y \le \dfrac{\pi}{2}$;

$u(x,\dfrac{\pi}{2}) = 1 + x^2$, $2 \le x \le 3$; $u(3,y) = 9 + \sin y$, $0 \le y \le \dfrac{\pi}{2}$.

15. $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 2x + e^y$;

$(x,y) \in \overline{G} = \{(x,y),\ 0 \le x \le 0.5,\ 0 \le y \le 1\}$;

$u(x,0) = \dfrac{1}{3}x^3 + 1$, $0 \le x \le 0.5$; $u(0,y) = e^y$, $0 \le y \le 1$;

$u(x,1) = \dfrac{1}{3}\left(x^3 + 8.154845\right)$, $0 \le x \le 0.5$; $u(0.5,y) = 0.041667 + e^y$, $0 \le y \le 1$.

16. $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 1 + \cos x - \sin y$;

$(x,y) \in \overline{G} = \left\{(x,y),\ 0 \le x \le \dfrac{\pi}{2},\ 0 \le y \le \pi\right\}$;

$u(x,0) = \dfrac{1}{2}x^2 - \cos x$, $0 \le x \le \dfrac{\pi}{2}$; $u(0,y) = \sin y - 1$, $0 \le y \le \pi$;

$u(x,\pi) = \dfrac{1}{2}x^2 - \cos x$, $0 \le x \le \dfrac{\pi}{2}$; $u\left(\dfrac{\pi}{2},y\right) = \sin y + 1.2337$, $0 \le y \le \pi$.

17. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=\left(3+y^2\right)e^x$;

$(x,y)\in\overline{G}=\left\{(x,y),\ \ 0\le x\le\dfrac{1}{2},\ \ 0\le y\le 1\right\}$;

$u(x,0)=e^x,\ 0\le x\le\dfrac{1}{2}$; $u(0,y)=1+y^2,\ 0\le y\le 1$;

$u(x,1)=2e^x,\ 0\le x\le\dfrac{1}{2}$; $u\left(0.5,y\right)=1.64872(1+y^2),\ 0\le y\le 1$.

18. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=2\left(x^2+y^2\right)$;

$(x,y)\in\overline{G}=\left\{(x,y),\ \ 0\le x\le 1,\ \ 0\le y\le 1\right\}$;

$u(x,0)=2,\ 0\le x\le 1$; $u(0,y)=2,\ 0\le y\le 1$;

$u(x,1)=2+x+x^2,\ 0\le x\le 1$; $u\left(1,y\right)=2+y+y^2,\ 0\le y\le 1$.

19. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=2+xe^y$; $(x,y)\in\overline{G}=\left\{(x,y),\ \ 2\le x\le 3,\ \ 0\le y\le 2\right\}$;

$u(x,0)=1+x+x^2,\ 2\le x\le 3$; $u(2,y)=5+2e^y,\ 0\le y\le 2$;

$u(x,2)=1+7.38906x+x^2,\ 2\le x\le 3$; $u\left(3,y\right)=10+3e^y,\ 0\le y\le 2$.

20. $\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=2+ye^x$; $(x,y)\in\overline{G}=\left\{(x,y),\ \ 1\le x\le 2,\ 2\le y\le 3\right\}$;

$u(x,2)=11+2e^x,\ 1\le x\le 2$; $u(1,y)=y^2+2.7183y+7,\ 2\le y\le 3$;

$u(x,3)=16+3e^x,\ 1\le x\le 2$; $u\left(2,y\right)=y^2+7.38906y+7,\ 2\le y\le 3$.

9 РЕШЕНИЕ ЗАДАЧ НЕСТАЦИОНАРНОЙ ТЕПЛОПРОВОДНОСТИ С ПОМОЩЬЮ ЯВНЫХ И НЕЯВНЫХ РАЗНОСТНЫХ СХЕМ

Задачи математического моделирования, использующие в своих постановках многомерные нестационарные уравнения, при численной реализации с помощью конечно-разностных методов требуют большого количества компьютерного времени. Одним из способов преодоления этой проблемы является использование многопроцессорных вычислительных систем. В данном разделе будут рассматриваться многопроцессорные вычислительные системы с распределенной памятью, так как в настоящее время большинство супер-ЭВМ, которыми мы располагаем, построены именно по такому принципу.

Наиболее общим подходом равномерного распределения вычислительной нагрузки между процессорными элементами при решении сеточных уравнений является распределение вычислительных областей на подобласти по принципу геометрической декомпозиции. Причем самым простым способом решения этих разделенных на подобласти конечно-разностных задач является применение явных разностных схем. Однако такие схемы отличаются медленной скоростью сходимости для стационарных краевых задач и малым шагом интегрирования – для нестационарных задач, ограничения на который накладывает условие устойчивости.

Неявные схемы относительно свободны от этого недостатка. Однако распараллеливание неявных схем с глобальной пространственной связанностью данных и необходимостью решения линейных систем большой размерности при параллельной реализации становится весьма нетривиальной задачей. Для решения системы линейных алгебраических уравнений необходимо выбрать алгоритм, который при распараллеливании сохранит свойства последовательного и покажет хорошие результаты по скорости вычислений (высокую эффективность). Конечно, решение СЛАУ требует больших вычислительных затрат, однако выигрыш проявляется в свойствах неявных схем. Как правило, неявные схемы абсолютно устойчивы, и поэтому шаг по времени можно взять довольно большой, в то время как при использовании явных схем встречаются серьезные ограничения на шаг по времени.

Распараллеливание и само знакомство с явными и неявными схемами будут осуществляться на примере численного решения уравнения теплопроводности в квадрате с граничными условиями первого рода и заданными начальными условиями. Задача состоит в следующем: найти распределение скалярной функции температуры $T(t,x,y)$ в течение выхода процесса передачи тепла теплопроводностью на стационарный режим.

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right); \tag{9.1}$$

$$\begin{cases} T\big|_G = T_b(x,y);\ T_{t=0} = 100; \\ 0 \le x, y \le 1;\ 0 \le t \le T. \end{cases} \tag{9.2}$$

Приближенное решение данной задачи будем искать с использованием метода конечных разностей. Для области исследования построим равномерную сетку

$$\varpi_h = \left\{ \begin{array}{l} x_i = h_x \cdot i;\ h_x = 1/(Nx+1);\ i = 0,...,Nx+1 \\ y_j = h_y \cdot j;\ h_y = 1/(Ny+1);\ j = 0,...,Ny+1 \end{array} \right\}.$$

Будут использоваться следующие обозначения: $T\left(t_n, x_i, y_j\right) = T^n_{i,j}$. Верхний индекс определяет принадлежность к временному слою, а нижние индексы – принадлежность к (i,j)-му узлу сетки.

Приведем конечно-разностные формулы для аппроксимации дифференциальных операторов:

$$\left(\frac{\partial T}{\partial t}\right)^n_{i,j} \approx \frac{T^{n+1}_{i,j} - T^n_{i,j}}{\tau};$$

$$\left(\frac{\partial^2 T}{\partial x^2}\right)^n_{i,j} \approx \lambda\left(\frac{T^{n+1}_{i+1,j} - 2\cdot T^{n+1}_{i,j} + T^{n+1}_{i-1,j}}{h_x^2}\right) + (1-\lambda)\left(\frac{T^n_{i+1,j} - 2\cdot T^n_{i,j} + T^n_{i-1,j}}{h_x^2}\right);$$

$$\left(\frac{\partial^2 T}{\partial y^2}\right)^n_{i,j} \approx \lambda\left(\frac{T^{n+1}_{i,j+1} - 2\cdot T^{n+1}_{i,j} + T^{n+1}_{i,j-1}}{h_y^2}\right) + (1-\lambda)\left(\frac{T^n_{i,j+1} - 2\cdot T^n_{i,j} + T^n_{i,j-1}}{h_y^2}\right).$$

Здесь λ – коэффициент, определяющий вид разностной схемы ($0 \le \lambda \le 1$), если $\lambda = 0$, то схема явная, если $\lambda = 1$, то схема неявная. Это два крайних случая, при $0 < \lambda < 1$ получаем схемы смешанного типа. Особого внимания заслуживает схема Кранка–Николсона, для которой $\lambda = 0.5$. Эта схема имеет второй порядок аппроксимации.

Применив эти конечно-разностные формулы к дифференциальному уравнению во внутренних узлах сетки, получим систему линейных алгебраических уравнений. Начальные и граничные условия на используемой равномерной сетке аппроксимируются точно.

$$\begin{cases} \dfrac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\tau} = \alpha \left(\begin{array}{l} \lambda \left(\dfrac{T_{i+1,j}^{n+1} - 2 \cdot T_{i,j}^{n+1} + T_{i-1,j}^{n+1}}{h_x^2} \right) + \\ (1-\lambda) \left(\dfrac{T_{i+1,j}^{n} - 2 \cdot T_{i,j}^{n} + T_{i-1,j}^{n}}{h_x^2} \right) + \\ \lambda \left(\dfrac{T_{i,j+1}^{n+1} - 2 \cdot T_{i,j}^{n+1} + T_{i,j-1}^{n+1}}{h_y^2} \right) + \\ (1-\lambda) \left(\dfrac{T_{i,j+1}^{n} - 2 \cdot T_{i,j}^{n} + T_{i,j-1}^{n}}{h_y^2} \right) \end{array} \right); \\ \qquad i = 1, Nx; \;\; j = 1, Ny; \; n = 0,1,2,... \\ T_{0,j}^{n+1} = T_b(x_0, y_j); \quad T_{Nx+1,j}^{n+1} = T_b(x_{Nx+1}, y_j); \quad j = 0, Ny+1; \; n = 0,1,2,... \\ T_{i,0}^{n+1} = T_b(x_i, y_0); \quad T_{i,Ny+1}^{n+1} = T_b(x_i, y_{Ny+1}); \quad i = 0, Nx+1; \; n = 0,1,2,... \\ T_{i,j}^{0} = 100; \quad i = 0, Nx+1; \;\; j = 0, Ny+1. \end{cases}$$

Далее необходимо определиться, какую схему будем применять для решения задачи (9.1) и (9.2). Если используется грубая сетка с большим шагом по координатам и ограничение на шаг по времени, накладываемое условием устойчивости, позволяет делать достаточно крупные шаги по времени, то выгодно применять явную разностную схему, которая более проста в своей реализации. Если же сетка мелкая и из условной устойчивости явных схем требуется небольшой шаг по времени, то выгодно применять неявные схемы.

Какую разностную схему получим после аппроксимации – явную, неявную или смешанную (явно-неявную), зависит от того, чему равен коэффициент λ при аппроксимации производных по пространству.

Рассмотрим вначале параллельную реализацию явной разностной схемы для решения уравнения теплопроводности.

9.1 Явная схема

В этом случае ($\lambda = 0$) для аппроксимации производных по пространству используются значения сеточной функции $T_{i,j}$ с n-го временного слоя. Тогда получим:

$$\begin{cases} T_{i,j}^{n+1} = T_{i,j}^{n} + \tau \cdot \alpha \left(\dfrac{T_{i+1,j}^{n} - 2 \cdot T_{i,j}^{n} + T_{i-1,j}^{n}}{h_x^2} \right) \\ \qquad + \tau \cdot \alpha \left(\dfrac{T_{i,j+1}^{n} - 2 \cdot T_{i,j}^{n} + T_{i,j-1}^{n}}{h_y^2} \right); \\ \qquad i = 1, Nx; \quad j = 1, Ny; \\ T_{0,j}^{n+1} = T_b\left(x_0, y_j\right); \quad T_{Nx+1,j}^{n+1} = T_b\left(x_{Nx+1}, y_j\right); \\ n = 0,1,2,...; \quad j = 0,..., Ny+1; \\ T_{i,0}^{n+1} = T_b\left(x_i, y_0\right); \quad T_{i,Ny+1}^{n+1} = T_b\left(x_i, y_{Ny+1}\right); \\ n = 0,1,2,...; \quad i = 0,..., Nx+1; \\ T_{i,j}^{0} = 100; \; i = 0,..., Nx+1; \quad j = 0,..., Ny+1. \end{cases} \tag{9.3}$$

Таким образом, в (9.3) представлена явная формула для вычислений значений сеточной функции на новом временном слое. Перепишем полученную формулу в следующем виде:

$$T_{i,j}^{n+1} = (1 + ap) T_{i,j}^{n} + ae \cdot T_{i+1,j}^{n} + aw \cdot T_{i-1,j}^{n} + an \cdot T_{i,j+1}^{n} + as \cdot T_{i,j-1}^{n}. \tag{9.4}$$

Коэффициенты ap, ae, aw, an, as легко определяются из формулы (9.3):

$$ae = \frac{\alpha \cdot \tau}{h_x^2};\ aw = \frac{\alpha \cdot \tau}{h_x^2};\ an = \frac{\alpha \cdot \tau}{h_y^2};\ as = \frac{\alpha \cdot \tau}{h_y^2};$$

$$ap = -(ae + aw + an + as).$$

Формула (9.4) легко программируется и очень проста для применения:

```
Do i=1,Nx
   Do j=1,Ny
      Tnew(i,j)=(1+ap)*T(i,j)+ae*T(i+1,j)+aw*T(i-1,j)
                             +an*T(i,j+1)+as*T(i,j-1)
   End do
End do
```

Далее обсудим возможности распараллеливания программы для нахождения приближенного решения поставленной задачи. В рассматриваемом примере возможны два различных способа разделения данных – *одномерная* или *ленточная* схема или *двухмерное* или *блочное* разбиение узлов вычислительной сетки. Из практики параллельных вычислений хорошо известно, что в целом более эффективная двумерная декомпозиция расчетной области теряет свои преимущества перед одномерной в том случае, если число процессоров параллельного компьютера невелико. Рассмотрим отдельно оба варианта декомпозиции данных.

9.1.1 Одномерная декомпозиция

Для определенности декомпозицию расчетной области проведем по индексу j (рис. 9.1), хотя на практике выбор координаты для разбиения определяется особенностями языка программирования или способом хранения данных в оперативной памяти. Совокупность узлов расчетной сетки, попавших на один процессорный элемент, будем называть полосой.

После того как данные были распределены, переходим к следующему этапу построения параллельной программы – проектированию коммуникаций. При аппроксимации дифференциальной задачи использовался шаблон «крест» (см. рис. 9.1), поэтому для организации вычислений в приграничных узлах сеточной подобласти

каждого процессорного элемента требуется получить векторы сеточных значений приближенного решения с предшествующей и последующей подобластей декомпозиции. Например, для области $R=1$ – с областей $R=0$ и $R=2$.

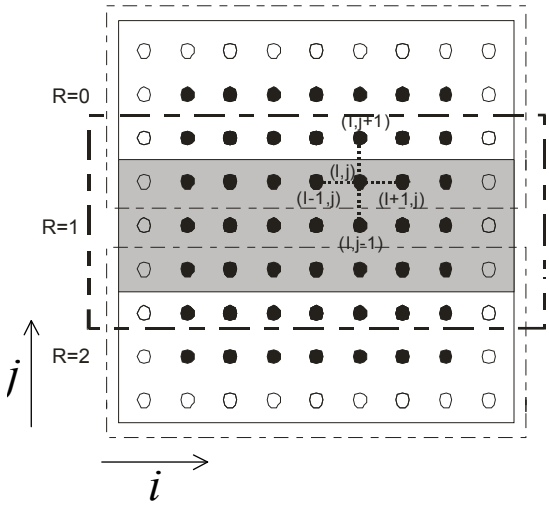

Рис. 9.1 Одномерная декомпозиция расчетной области (открытые кружки представляют собой граничные узлы сетки)

Переданные другими процессорными элементами векторы используются только при проведении расчетов во внутренних узлах сетки сеточной подобласти. Расчет же значений сеточной функции, объединенных в пересылаемые векторы, производится в сеточных подобластях своего исходного месторасположения. Поэтому пересылки граничных значений сеточной функции должны осуществляться перед началом выполнения каждой очередной глобальной итерации метода.

Процедура обмена сеточными значениями искомой функции между соседними (по направлению декомпозиции расчетной области) процессорными элементами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессорному элементу и принимает верхнюю граничную строку от предыдущего процессорного эллемента (рис. 9.2). Вторая часть передачи данных выполняется в обратном направлении: процессоры пере-

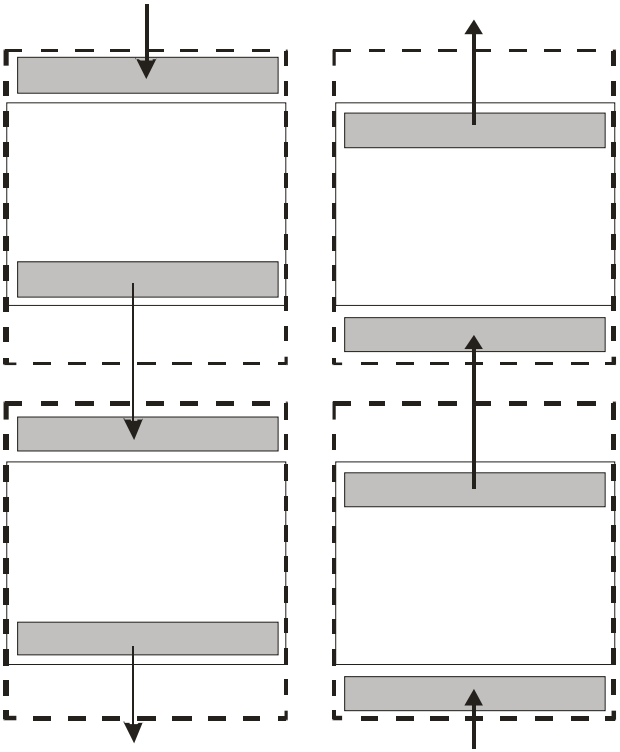

Рис. 9.2 Схема обменов между вычислительными узлами

дают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки отследующих процессорных элементов.

9.1.2 Двумерная декомпозиция

При большом числе используемых процессоров перспективнее становится двумерная декомпозиция. При этом по-прежнему необходимо в адресном пространстве каждого процессорного элемента создавать дополнительные фиктивные ячейки для обеспечения однородности вычислений. Операции обменов схематически изображены на рис. 9.3. На первый взгляд может показаться, что обменов стало больше. Проверим, что происходит на самом деле. Посчитав число одновременно пересылаемых элементов при каждом способе

декомпозиции, заметим, что в 1d-декомпозиции количество пересылаемых данных есть величина порядка $2n$ (n – размерность задачи, т.е. количество узлов сетки по каждому координатному направлению), а в 2d-декомпозиции количество пересылаемых данных есть величина порядка $\frac{4n}{\sqrt{p}}$, где p – число используемых процессоров. Оценки приведены для решения двумерной задачи.

Таблица 9.1 Количество пересылаемых данных при различных способах декомпозиции и различном числе процессоров

Число процессоров	2	4	8	16
1d-декомпозиция	$2 \cdot n$	$2 \cdot n$	$2 \cdot n$	$2 \cdot n$
2d-декомпозиция	$\frac{4 \cdot n}{\sqrt{2}}$	$2 \cdot n$	$\frac{2 \cdot n}{\sqrt{2}}$	n

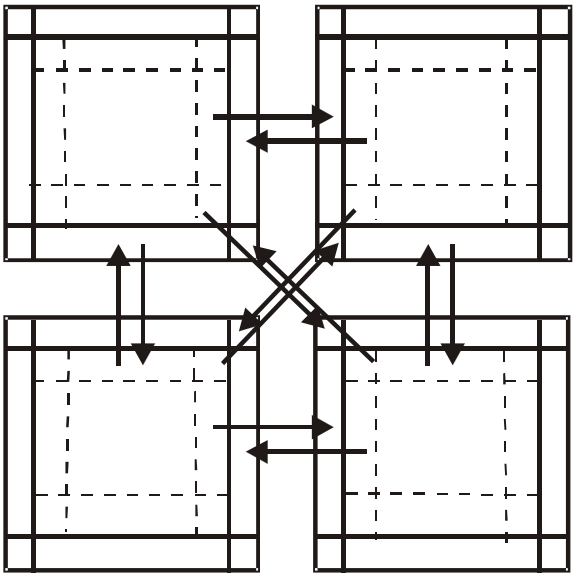

Рис. 9.3 Схема обменов при двумерной декомпозиции

Таким образом, из табл. 9.1 видно, что при $p > 4$ двумерная декомпозиция является более перспективной для распараллеливания. В то же время реализация 2d-декомпозиции является более трудоемким (с точки зрения программирования) процессом по сравнению с реализацией 1d-декомпозиции. Основная возникающая сложность состоит в организации межпроцессорных обменов. Есть различные способы решения этой проблемы. Ниже представлен вариант с использованием декартовой топологии и новых типов для организации обменов, а также вариант с перенумерацией неизвестных значений сеточной функции для сбора решения на одном процессорном элементе. Эти два способа организации обменов являются взаимозаменяемыми.

В заключение необходимо обратить внимание, что все характеристики параллельного алгоритма (ускорение, время пересылок) являются относительными и принимают по-настоящему решающее значение лишь при рассмотрении конкретных задач. Для каждой задачи нужно выбирать свой метод распараллеливания, свою декомпозицию, руководствуясь при этом общепринятыми приемами и рекомендациями параллельного программирования.

9.2 Неявная схема

В этом случае ($\lambda = 1$) для аппроксимации производных по пространству используются значения сеточной функции $T_{i,j}$ с $n+1$-го временного слоя. Тогда получаем:

$$
\begin{cases}
T_{i,j}^{n+1} = T_{i,j}^{n} + \tau \cdot \alpha \left(\dfrac{T_{i+1,j}^{n+1} - 2 \cdot T_{i,j}^{n+1} + T_{i-1,j}^{n+1}}{h_x^2} \right) \\
\qquad\qquad + \tau \cdot \alpha \left(\dfrac{T_{i,j+1}^{n+1} - 2 \cdot T_{i,j}^{n+1} + T_{i,j-1}^{n+1}}{h_y^2} \right); \\
\qquad i = 1, Nx; \quad j = 1, Ny; \\
T_{0,j}^{n+1} = T_b\left(x_0, y_j\right); \qquad T_{Nx+1,j}^{n+1} = T_b\left(x_{Nx+1}, y_j\right); \\
n = 0,1,2,...; \; j = 0,...,Ny+1; \\
T_{i,0}^{n+1} = T_b\left(x_i, y_0\right); \qquad T_{i,Ny+1}^{n+1} = T_b\left(x_i, y_{Ny+1}\right); \\
n = 0,1,2,...; \; i = 0,...,Nx+1; \\
T_{i,j}^{0} = 100; \; i = 0,...,Nx+1; \quad j = 0,...,Ny+1.
\end{cases}
\tag{9.5}
$$

Для большей наглядности перепишем полученные формулы в следующем виде:

$$(1-ap)T_{i,j}^{n+1} = ae \cdot T_{i+1,j}^{n+1} + aw \cdot T_{i-1,j}^{n+1} + an \cdot T_{i,j+1}^{n+1} + as \cdot T_{i,j-1}^{n+1} + b_{i,j}. \quad (9.6)$$

Коэффициенты $ap, ae, aw, an, as, b_{i,j}$ легко определяются из формулы (9.5):

$$ae = \frac{\alpha \cdot \tau}{h_x^2};\ aw = \frac{\alpha \cdot \tau}{h_x^2};\ an = \frac{\alpha \cdot \tau}{h_y^2};\ as = \frac{\alpha \cdot \tau}{h_y^2};\ b_{i,j} = T_{i,j}^n$$

$$ap = -\left(ae + aw + an + as\right);\ \ i = 1, Nx;\ \ j = 1, Ny;$$

$$i = 0; \qquad ap = 1;\ \ b_{i,j} = T_b(x_0, y_j); \qquad ae = aw = as = an = 0;$$

$$i = Nx+1;\ \ ap = 1;\ \ b_{i,j} = T_b(x_{Nx+1}, y_j);\ \ ae = aw = as = an = 0;$$

$$j = 0; \qquad ap = 1;\ \ b_{i,j} = T_b(x_i, y_0); \qquad ae = aw = as = an = 0;$$

$$j = Ny+1;\ \ ap = 1;\ \ b_{i,j} = T_b(x_i, y_{Ny+1});\ \ ae = aw = as = an = 0.$$

В случае неявной разностной схемы вместо проведения вычислений по готовой явной формуле необходимо предварительно выполнить решение системы линейных алгебраических уравнений большой размерности с разреженной матрицей. Преимущества этого подхода были описаны выше, теперь непосредственно перейдем к выбору метода решения системы линейных алгебраических уравнений.

Прямые методы, как правило, не используются для решения систем линейных алгебраических уравнений вида (9.5), полученных после аппроксимации дифференциальной задачи, так как их применение имеет ряд существенных недостатков. Например, необходимо целиком хранить матрицу в оперативной памяти, что делает ее использование нерациональным. Эти методы не уменьшают влияния погрешности округления, что становится неприемлемым при решении плохо обусловленных задач большой размерности. Поэтому целесообразно использовать итерационные методы решения СЛАУ. Например, метод Зейделя или метод сопряженных градиентов.

9.2.1 Метод сопряженных градиентов

Одними из перспективных методов итерационного решения систем вида $Ax = b$ являются алгоритмы, построенные на основе выбора итерационных параметров из условия минимизации функционалов, определяющих точность текущих последовательных приближений.

В дальнейшем будут использованы следующие обозначения: $r^k = b - Ax^k$ – невязка; $\omega^k = B^{-1}r^k$ – поправка и $z^k = x^k - x$ – погрешность (ошибка). Здесь x – точное решение системы $Ax = b$; x^k – k-е приближение к точному решению; B – предобуславливающая матрица.

К открытию метода сопряженных градиентов (CG) независимо пришли М. Хестенес и Э. Штифель. Он является наиболее предпочтительным по быстродействию для симметричных положительно определенных систем. Формулы классического метода сопряженных градиентов имеют следующий вид:

$p = x;$
$v = Ap;$
$p = b - v;$
$r = p;$
$\alpha = \left(\|r\|_2\right)^2;$
for $\left((i < i\_\max).and.(\alpha > \varepsilon) \right)$
$v = Ap;$
$\lambda = \alpha \Big/ (v, p)_2;$
$x = x + \lambda p;$
$r = r - \lambda v;$
$\alpha_{new} = \left(\|r\|_2\right)^2;$
$p = r + \left(\alpha_{new} \Big/ \alpha \right) p;$
$\alpha = \alpha_{new};$
end do.

Достаточным условием сходимости метода сопряженных градиентов являются симметричность и положительная определенность матрицы A, при этом если для спектра матрицы выполняется условие $0 < m \le \gamma(A) \le M$, то скорость сходимости можно определить по формуле

$$\phi(x^n) \le \left(\frac{2\cdot\gamma^n}{1+\gamma^{2n}}\right)\phi(x^0);$$

$$\gamma = \frac{1-\sqrt{m/M}}{1+\sqrt{m/M}}.$$

Скорость сходимости метода сопряженных градиентов выше, чем скорость сходимости метода Зейделя.

Метод CG имеет то особенно привлекательное свойство, что в его реализации предусмотрено одновременное хранение в памяти лишь четырех векторов. Кроме того, в его внутреннем цикле помимо матрично-векторного произведения вычисляются только два скалярных произведения, три операции типа «saxpy» (сложение вектора, умноженного на число, с другим вектором) и небольшое количество скалярных операций. Таким образом, и необходимая оперативная память, и объем вычислительной работы в методе не очень велики. В CG алгоритме x^{k+1} вычисляется с использованием рекуррентных соотношений для трех групп векторов. Одновременно в памяти требуется хранить лишь самые последние векторы из каждой группы, которые переписываются поверх ранее рассчитанных значений. Первая группа векторов – это приближенные решения x^k. Вторая группа – это невязки $r^k = b - Ax^k$. Третья группа – это сопряженные градиенты p^k. Эти векторы называют градиентами по следующей причине: шаг метода CG можно рассматривать как выбор числа a_{new}/a из условия, чтобы новое приближенное решение $x^{k+!} = x^k + a_{new}/a \cdot p^k$ минимизировало норму невязки $\|r_k\|_{A^{-1}} = \left(r_k^T A^{-1} r_k\right)^{1/2}$. Иными словами, p^k используется как направление градиентного поиска. Они называются сопряженными или, точнее, A-сопряженными, потому что $p_k^T A p_j = 0$ при $j \ne k$.

Параллельную реализацию метода сопряженных градиентов рассмотрим на примере приближенного решения уравнения теплопро-

водности с помощью неявных разностных схем. Чтобы яснее представить себе процесс распараллеливания и связанные с ним изменения в алгоритме, разберем отдельно каждую матрично-векторную операцию, которая используется в изложенном выше методе.

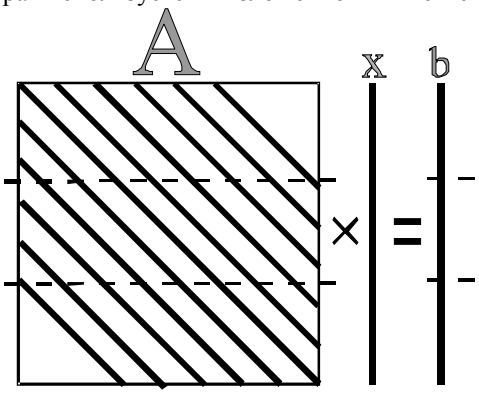

Рис. 9.4 Строчное распределение

Первая операция – это матрично-векторное произведение $y = Ax$. Известно, что в общем случае при умножении матрицы на вектор возможны два способа распределения данных. В первом случае каждому процессорному элементу назначается определенное количество строк матрицы A и целиком вектор x (рис. 9.4). Каждый процессорный элемент реализует умножение распределенных ему строк матрицы A на вектор x (число вычислительных узлов меньше или равно числу строк). Получаем абсолютно не связанные подзадачи, однако предложенный алгоритм не лишен и недостатков. Дело в том, что компоненты результирующего вектора y оказываются разбросанными по всем процессорным элементам и для старта следующей итерации необходимо собрать вектор y на каждом процессорном элементе.

Другой вариант – каждому процессорному элементу распределяется определенное количество столбцов матрицы A, тогда отпадает необходимость хранить на каждом процессоре целиком вектор x, и он распределяется по всем вычислительным узлам (схема распределения данных представлена на рис. 9.5). Тогда в процессе вычислений результирующий вектор y оказывается разобранным на слагаемые, а каждое слагаемое находится на отдельном процессорном элементе. То есть опять потребуются пересылки данных для того,

чтобы собрать результирующий вектор y и разослать каждому процессорному элементу часть вектора y, необходимую для начала следующей итерации.

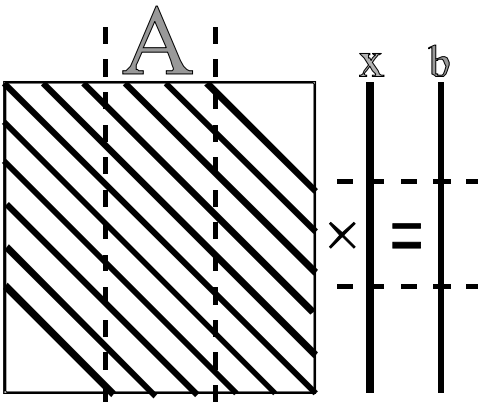

Рис. 9.5 Столбцовое распределение

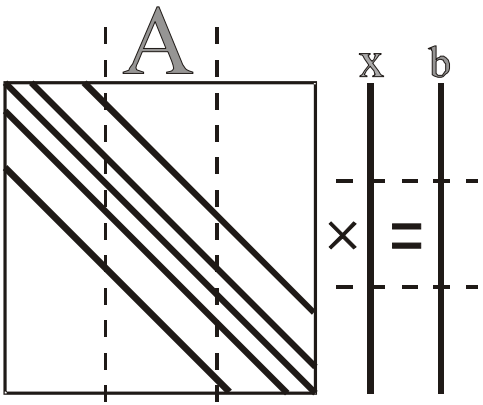

Рис. 9.6 Случай разреженной матрицы

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 y, часто будут отличны от нуля лишь

на одном процессорном элементе, т.е. сокращается количество межпроцессорных обменов.

Следующие две операции – это вычисление скалярного произведения и векторная операция saxpy. При 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.

Вычисление скалярного произведения – принципиально другая операция, ее результатом является не вектор, а одно число, которое впоследствии должно быть разослано на все вычислительные узлы. 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 полученного вектора, затем результаты сложения собираются и суммируются на одном из вычислительных узлов, полученный результат рассылается остальным вычислительным узлам.

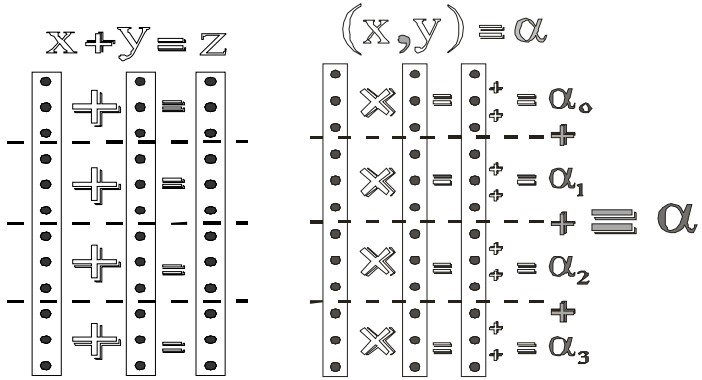

Рис. 9.7 Схема операций saxpy и скалярного произведения

Тестовые расчеты, проведенные для различных значений коэффициента температуропроводности, показали незначительное увеличение времени счета с ростом α . Это объясняется сокращением числа итераций, требуемых для сходимости итерационного метода сопряженных градиентов на каждом временном шаге. Рисунок 9.8

показывает, сколько итераций метода необходимо для сходимости вычислительного процесса на каждом временном шаге. Кроме того, расчетным путем было показано, что применение неявной схемы при больших значениях параметра α имеет преимущество, поскольку не требует существенного ограничения величины шага по времени по сравнению с явными разностными схемами.

Рис. 9.8 Количество локальных итераций метода сопряженных градиентов на каждом шаге по времени

Таблица 9.2 Время работы параллельной программы решения уравнения теплопроводности с помощью неявной и явной разностных схем, с

Число процессорных элементов	1	4	9
Неявная схема	112	56	36
Явная схема	318	89	48

В табл. 9.2 представлено время выполнения параллельных программ, использующих одномерную декомпозицию, при решении нестационарного уравнения теплопроводности с помощью явной и неявной разностных схем. Для принятого значения коэффициента температуропроводности $\alpha = 10^{-4}$ 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 неявная схема имеет преимущество, которое, однако, уменьшается с ростом числа используемых процессорных элементов, что связано с увеличением затрат на межпроцессорную передачу данных.

```
      PROGRAM Example1
c  Решение двумерного уравнения теплопроводности в
c  единичном  квадрате с использованием явной разностной
c схемы. 1d-декомпозиция
      Implicit none
      Include 'mpif.h'
c  Параметры:
c  Nx,Ny - количество точек области в каждом направлении,
c  m - количество точек на процессор
c  T,Tnew - численное решение задачи
      Integer i, j, Ny, Nx, m
      Integer comm, ierr, size, rank, left, right, tag, status(100)
      Parameter (Nx=720, Ny=Nx)
      Double precision ap, ae, aw, an, as, hx, hy, Lx, Ly, Tpas(Nx,Ny),
     1              T(0:Nx+1,0:Ny+1), Tnew(0:Nx+1,0:Ny+1), alfa,
     2              tau, time, timefin, timeStart, timeStop,
     3              timeStart1, timeStop1
      Parameter (timefin=1.0d3, Lx=1.0d0, Ly=1.0d0,
     1         hx=Lx/(Nx+1), hy=Ly/(Ny+1), alfa=1.0d-5)

c  Инициализация, определение числа выделенных процессоров
c  (size)  и номера текущего процесса (rank)
      Call MPI_INIT(ierr)
      Call MPI_comm_size(mpi_comm_world, size, ierr)
      Call MPI_comm_rank(mpi_comm_world, rank, ierr)
      comm = mpi_comm_world
c  Определение номеров процессоров (соседей) сверху и снизу.
c  Если таковые отсутствуют, то им присваивается значение
c  MPI_PROC_NULL (для них коммуникационные операции
c  игнорируются)
      If (rank.eq.0) then
        Left=MPI_PROC_NULL
      Else
        Left=rank-1
      End if
      If (rank.eq.size-1) then
        right=MPI_PROC_NULL
      Else
        right=rank+1
      End if
```

```
c  Определение числа элементов, обрабатываемых одним
c  процессорным элементом
      m=Ny/size
      If (rank.lt.(Nx-size*m)) then
        m=m+1
      End if
c  Задание начальных значений
      Do i=0,Nx+1
        Do j=0,m+1
          T(i,j)=100.0d0
          Tnew(i,j)=100.0d0
        End do
      End do
c  Задание граничных условий
      If (rank.eq.0) then
        Do i=0,Nx+1
          T(i,0)=10.0d0
        End do
      End if
      If (rank.eq.size-1) then
        Do i=0,Nx+1
          T(i,m+1)=10.0d0
        End do
      End if
      Do j=0,m+1
        T(0,j)=1.0d0
        T(Nx+1,j)=10.0d0
      End do
c  Подготовка к расчетам
      tau = 0.2*(hx*hx*hy*hy)/(alfa*(hx**2+hy**2))
      ae=alfa*tau/(hx*hx)
      aw=alfa*tau/(hx*hx)
      an=alfa*tau/(hy*hy)
      as=alfa*tau/(hy*hy)
      ap=-(ae+aw+an+as)
      time = 0
      If (rank.eq.0) then
        open (2,file='rezult1.dat')
        timeStart = MPI_Wtime()
        timeStop1=0.0
```

```
      End if
c  Движение по временной оси
      Do while (time.lt.timefin)
c  Расчет значений с нового временного слоя
        Do j=1,m
          Do i=1,Nx
            Tnew(i,j) = T(i,j)*(1+ap) + T(i+1,j)*ae + T(i-1,j)*aw
     1                                + T(i,j+1)*an + T(i,j-1)*as
          End do
        End do
        time =time + tau
c  Переприсваивание во внутренних узлах области
        Do j=1,m
          Do i=1,Nx
            T(i,j)=Tnew(i,j)
          End do
        End do
c  Пересылка граничных значений на соседние ПЭ
        tag =100
        If (rank.eq.0) timeStart1 = MPI_Wtime()
        Call MPI_SENDRECV(T(0,1), Nx+2, MPI_double_precision,
     1                    left, tag,
     2                    T(0,0), Nx+2, MPI_double_precision,
     3                    left, tag, comm, status, ierr)
        Call MPI_SENDRECV (T(0,m), Nx+2, MPI_double_precision,
     1                     right, tag,
     2                     T(0,m+1), Nx+2, MPI_double_precision,
     3                     right, tag, comm, status, ierr)
        If (rank.eq.0) timeStop1= timeStop1+MPI_Wtime()-timeStart1
      End do
c  Фиксация времени окончания расчетов
      If (rank.eq.0) timeStop=MPI_Wtime()-timeStart
      Do i=1,Nx
        Do j=1,Ny
          Tpas(i,j)=T(i,j)
        End do
      End do
c  Сбор результатов на нулевом процессорном элементе
      Call MPI_GATHER( Tpas(1,1), m*Nx,
     1                          MPI_DOUBLE_PRECISION,
```

```
     2                         Tpas(1,1), m*Nx,
     3                         MPI_DOUBLE_PRECISION,
     4                         0, comm, ierr)
      If (rank.eq.0) then
       Do i=1,Nx
        Do j=1,Ny
         T(i,j)=Tpas(i,j)
         T(0,j)=1.0
         T(Nx+1,j)=10.0
         T(i,0)=10.0
         T(i,Ny+1)=10.0
        End do
       End do
       Do i=0,Nx+1
        write (2,'(760f15.4)') (T(i,j),j=0,Ny+1)
       End do
       Write (2,'(a,i15,a,f15.9)')     'Size=',size,'  TimeC=',TimeStop
       Write (2,'(a,f15.9,a,f15.9)') ' TimeF=',time, ' tau=',tau
       Write (2,'(a,f15.9,a,f15.9)') '  Tcomp=',TimeStop-TimeStop1,
     1                                '  Tcomm=',TimeStop1
       Write(2,*) '  T(Nx/2,Ny/2)= ', T(Nx/2,Ny/2)
       Close(2)
      End if
      Call MPI_FINALIZE(ierr)
      End
```

```
      PROGRAM Example2
c  Решение двумерного уравнения теплопроводности в
c  единичном квадрате с использованием явной разностной
c  схемы. 2d-декомпозиция
      Implicit none
      Include 'mpif.h'
c  Параметры:
c  Nx,Ny - количество точек области в каждом направлении,
c  idim, kdim - количество процессоров в каждом направлении,
c  T,Tnew - численное решение задачи
      Integer i, j, l, ki, kj, Ny, Nx, idim, kdim, is, ks, icoor,kcoor
      Integer comm, ierr, size, rank, tag, status(100)
      Integer left, right, lower, upper,
     1        vertical_border, horizontal_border
```

```
      Integer dims(1:2), coords(1:2)
      Parameter (Nx=720, Ny=Nx)
      Double precision ap, ae, aw, an, as, hx, hy, Lx, Ly,
     1                 T(0:Nx+1,0:Ny+1), Tnew(0:Nx+1,0:Ny+1), alfa,
     2                 tau, time, timefin, timeStart, timeStop,
     3                 timeStart1, timeStop1, timeStart2, timeStop2,
     4                 Tpas(0:(Nx+2)*(Ny+2))
      Logical period(1:2)
      Parameter (timefin=1.0d3, Lx=1.0d0, Ly=1.0d0,
     1          hx=Lx/(Nx+1), hy=Ly/(Ny+1), alfa=1.0d-5)
c  Инициализация, определение числа выделенных процессоров
c  (size) и номера текущего процесса (rank)
      Call MPI_INIT(ierr)
      Call MPI_comm_size(mpi_comm_world, size, ierr)
      Call MPI_comm_rank(mpi_comm_world, rank, ierr)
c  Определение числа элементов, обрабатываемых одним
c  процессорным элементом
      idim = int(sqrt(real(size)))
      kdim = int(sqrt(real(size)))
      If ( (idim*kdim).ne.(size) ) then
        Idim = int(sqrt(real(size/2)))
        Kdim = 2*idim
      End if
      If ( (idim*kdim).ne.(size) ) then
        If (rank.eq.0) Write (*,*) 'Incorrect dimensions'
        Stop 'Program terminated'
      End if
      dims(1) = idim
      dims(2) = kdim
      period(1) = .false.
      period(2) = .false.
      call MPI_CART_CREATE(mpi_comm_world,2,dims,period,
     1                     .true.,comm,ierr)
      call MPI_COMM_RANK(comm, rank, ierr)
      call MPI_CART_COORDS(comm, rank, 2, coords, ierr)
      icoor=coords(1)
      kcoor=coords(2)
      is=Nx/idim
      ks=Ny/kdim
      Call MPI_CART_SHIFT(comm, 0, 1, left, right, ierr)
```

```
      Call MPI_CART_SHIFT(comm, 1, 1, lower, upper, ierr)

      call MPI_TYPE_VECTOR(ks+2,1,Nx+2,
     1              MPI_DOUBLE_PRECISION,vertical_border, ierr)
      call MPI_TYPE_COMMIT(vertical_border, ierr)
      call MPI_TYPE_VECTOR(1,is+2,is+2,
     1              MPI_DOUBLE_PRECISION, horizontal_border,ierr)
      call MPI_TYPE_COMMIT(horizontal_border,ierr)
c  Задание начальных значений
      Do i=0,is+1
       Do j=0,ks+1
        T(i,j)=100.0d0
        Tnew(i,j)=100.0d0
       End do
      End do
c  Задание граничных условий
      If (icoor.eq.0) then
       Do j=0,ks+1
        T(0,j)=1.0d0
       End do
       If (kcoor.eq.0) then
        Do j=0,Ny+1
         T(0,j)=1.0d0
        End do
        Do i=0,Nx+1
         T(i,0)=10.0d0
        End do
       End if
      End if
      If (icoor.eq.idim-1) then
       Do j=0,ks+1
        T(is+1,j)=10.0d0
       End do
      End if
      If (kcoor.eq.0) then
       Do i=0,is+1
        T(i,0)=10.0d0
       End do
      End if
      If (kcoor.eq.kdim-1) then
```

```
        Do i=0,is+1
          T(i,ks+1)=10.0d0
        End do
      End if
c  Подготовка к расчетам
      tau = 0.2*(hx*hx*hy*hy)/(alfa*(hx**2+hy**2))
      ae=alfa*tau/(hx*hx)
      aw=alfa*tau/(hx*hx)
      an=alfa*tau/(hy*hy)
      as=alfa*tau/(hy*hy)
      ap=ae+aw+an+as
      time = 0
      If (rank.eq.0) then
        open (2,file='rezult1.dat')
        timeStart = MPI_Wtime()
        timeStop1=0.0
        timeStop2=0.0
      End if
c  Движение по временной оси
      Do while (time.lt.timefin)
       If (rank.eq.0) timeStart1 = MPI_Wtime()
c  Расчет значений с нового временного слоя
        Do j=1,ks
          Do i=1,is
            Tnew(i,j) = T(i,j)*(1-ap) + T(i+1,j)*ae + T(i-1,j)*aw
     1                                + T(i,j+1)*an + T(i,j-1)*as
          End do
        End do
        time =time + tau
c  Переприсваивание внутренних узлов области
        Do j=1,ks
          Do i=1,is
            T(i,j)=Tnew(i,j)
          End do
        End do
c  Пересылка граничных значений на соседние ПЭ
        tag =100
        If (rank.eq.0) timeStop1=timeStop1+MPI_Wtime()-timeStart1
        If (rank.eq.0) timeStart2 = MPI_Wtime()
        tag=10
```

```
      Call MPI_SENDRECV (T(0,ks),1,horizontal_border,upper,tag,
     1                   T(0,ks+1),1,horizontal_border,upper,
     2                   tag,comm,status,ierr)
      Call MPI_SENDRECV (T(0,1),1,horizontal_border,lower,tag,
     1                   T(0,0),1,horizontal_border,lower,
     2                   tag,comm,status,ierr)
      Call MPI_SENDRECV (T(is,0),1,vertical_border,right,tag,
     1                   T(is+1,0),1,vertical_border,right,
     2                   tag,comm,status,ierr)
      Call MPI_SENDRECV (T(1,0),1,vertical_border,left,tag,
     1                   T(0,0),1,vertical_border,left,
     2                   tag,comm,status,ierr)
      If (rank.eq.0) timeStop2=timeStop2+MPI_Wtime()-timeStart2
     End do
c  Фиксация времени окончания расчетов
      If (rank.eq.0) timeStop=MPI_Wtime()-timeStart
      l=0  !!!   Организация сбора решения на одном процессоре
      Do i=1,is
       Do j=1,ks
        Tpas(l)=T(i,j)
        l=l+1
       End do
      End do
c  Сбор результатов на нулевом процессорном элементе
      Call MPI_GATHER(Tpas(0),(ks)*(is),
     1                MPI_DOUBLE_PRECISION,
     2                Tpas(0),(is)*(ks),
     3                MPI_DOUBLE_PRECISION,
     4                0,comm,ierr)
      If (rank.eq.0) then
       l=0
       Do ki=1,idim
        Do kj=1,kdim
         Do i=1,is
          Do j=1,ks
           T(i+(ki-1)*is,j+(kj-1)*ks)=Tpas(l)
           l=l+1
          End do
         End do
        End do
```

```
      End do
      Do i=0,Nx+1
        write (2,'(560f15.4)') (T(i,j),j=0,Ny+1)
      End do
      Write (2,'(a,2i15,a,f15.9)') ' Size=',idim,kdim,' TimeC= ',
     1                              TimeStop
      Write (2,'(a,f15.9,a,f15.9)')' TimeF=', time, ' tau=',tau
      Write (2,'(a,f15.9,a,f15.9)')'Tcomp=',TimeStop1,
     1                              ' Tcomm=',TimeStop2
       Write(2,*) 'n=',Nx,' T(Nx/2,Ny/2)=', T(Nx/2,Ny/2)
       Close(2)
     End if
     Call MPI_FINALIZE(ierr)
    End
```

9.3 Задания

1. Апельсины в течение короткого времени могут выдерживать отрицательные температуры. Предположим, что апельсин диаметром 0,1 м (λ=0,47 Вт/(м·град), c=3800 Дж/(кг·град), ρ=940 кг/м^3) имеет начальную температуру +5 °С. Температура воздуха внезапно падает до –5 °С. Построить математическую модель для определения момента времени, когда температура поверхности апельсина достигнет 0 °С. Коэффициент теплоотдачи от апельсина к воздуху равен 10 Вт/(м^2·град). Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различной размерности задачи и числа используемых процессов.

2. Начальная температура хлорвинилового шарика (λ=0,15 Вт/(м·град), α=8·10^{-8} м^2/с) диаметром 5 см равна 90 °С. Он погружается в бак с водой, имеющей температуру 20 °С. Коэффициент теплоотдачи от шарика в воде 20 Вт/(м^2·град). Построить математическую модель процесса охлаждения шарика. Найти время пребывания шарика в воде, по истечении которого температура в его центре достигнет 40 °С. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различных размеров вычислительной сетки и числа используемых процессов.

3. Длинный алюминиевый (λ=236 Вт/(м·град), $\alpha = 10^{-4}$ м^2/с) цилиндр диаметром 0,6 м имеет начальную температуру 200 °С. Его внезапно помещают в среду с температурой 70 °С и коэффициентом теплоотдачи 85 Вт/(м^2·град). Построить математическую модель процесса охлаждения цилиндра. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различных размеров сетки и числа используемых процессов.

4. Лист оконного стекла имеет толщину 4 мм. Температура одной поверхности 0 °С, а другой – +20 °С. Найти распределение температуры по толщине стекла через заданное время, если температура за окном упала до –10 °С. Записать математическую постановку задачи. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различной плотности узлов вычислительной сетки и числа используемых процессов.

5. Стенка большой печи толщиной 1,5 см изготовлена из чугуна (λ=83,5 Вт/(м·град), $\alpha = 22 \cdot 10^{-6}$ м^2/с). Температура горячего газа 1100 °С. Коэффициент конвективной теплоотдачи на внутренней поверхности стенки 250 Вт/(м^2·град). Наружная поверхность печи окружена воздухом с температурой 30 °С и коэффициентом теплоотдачи 20 Вт/(м^2·град)°С. Записать постановку задачи теплопроводности для стенки и найти распределение температур в ней. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различного числа используемых процессов.

6. Определить конечное распределение температуры в обручальном кольце с радиусами R=2,0 см и r=1,8 см, если температура в помещении 20 °С, а температуру тела человека можно принять равной 36,6 °С. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различных размеров вычислительной сетки и числа используемых процессов. $\alpha = 126 \cdot 10^{-6}$ м^2/с.

7. Бетонный цилиндр диаметром 10 см и длиной 2,5 м имеет начальную температуру 90 °С. Он охлаждается в воздухе при температуре 10 °С. Коэффициент теплоотдачи от бетона к воздуху 18 Вт/(м^2·град). Найти распределение температур в цилиндре с течением времени. Определить время, за которое температура в центре цилиндра достигнет 30 °С. Разработать численный метод реше-

ния задачи. Написать параллельную программу и провести ее тестирование.

8. Нужно нагреть кусок алюминиевой проволоки, пропуская по ней электрический ток. Диаметр проволоки 1 мм, длина 10 см, электрическое сопротивление 0,2 Ом. По ней пропускается постоянный ток силой 1 А в течение 60 с. Начальная температура проволоки 25 °С. Проволока находится в воздухе с температурой 25 °С и коэффициентом теплоотдачи 20 Вт/(м^2·град). Найти распределение температуры в проволоке с течением времени. Записать математическую формулировку задачи. Пояснение: объемная интенсивность внутренних источников тепловыделения при пропускании электрического тока = сила тока·сила тока·омическое сопротивление/объем. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование.

9. Электродетонатор имеет форму цилиндра диаметром 0,1 мм и длиной 5 мм. Он находится в воздухе с температурой 30 °С и коэффициентом теплоотдачи 10 Вт/(м^2·град). Теплофизические свойства детонатора: λ=20 Вт/(м·град), α =5·10^{-5} м^2/с, электрическое сопротивление 0,2 Ом. Пренебрегая излучением и утечками тепла в креплениях на концах детонатора, определить время, по истечении которого детонатор взорвется, если по детонатору пропускать постоянный ток силой 3 А. Пояснение: объемная интенсивность внутренних источников тепловыделения при пропускании электрического тока = сила тока·сила тока·омическое сопротивление/объем. Температура плавления материала детонатора 900 ^{o}С. Записать математическую постановку задачи. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование для различной размерности задачи и числа используемых процессов.

10. Начальная температура длинной стальной балки с сечением квадрата единичной длины равна 40 °С. Определить распределение температуры тела через 100 с, если на боковых границах задать следующие граничные условия: при X=0:T=20; при X=1:T=10; при Y=0:T=10; при Y=1:T=10. Записать математическую постановку задачи. Разработать численный метод решения задачи. Написать параллельную программу и провести ее тестирование.

Литература

1. *Старченко А.В., Есаулов А.О.* Параллельные вычисления на многопроцессорных вычислительных системах. – Томск: Изд-во Том. ун-та, 2002. – 56 с.
2. http://intel.ru
3. http://www.parallel.ru/cluster/intel_compilers.html
4. http://www.bytemag.ru/?ID=604337
5. http://ru.wikipedia.org
6. *Антонов А.С.* Параллельное программирование с использованием MPI. – М.: Изд-во МГУ, 2004. – 71 с.
7. *Букатов А.А., Дацюк В.Н., Жегуло А.И.* Программирование многопроцессорных вычислительных систем. – Ростов н/Д: Изд-во ООО «ЦВВР», 2003. – 208 с.
8. *Немнюгин С.А., Стесик О.Л.* Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 2002. – 200 с.
9. *Копченова Н.В., Марон И.А.* Вычислительная математика в примерах и задачах. – М.: Наука, 1972. – 367 с.
10. *Голуб Дж., Ван Лоун Дж.* Матричные вычисления. – М.: Мир, 1999. – 586 с.
11. *Деммель Дж.* Вычислительная линейная алгебра. – М.: Мир, 2001. – 430 с.
12. *Касьянов В.Н., Сабельфельд В.К.* Сборник заданий по практикуму на ЭВМ. – М.: Наука, 1986. – 272 с.
13. *Гергель В.П., Стронгин Р.Г.* Основы параллельных вычислений для многопроцессорных вычислительных машин. – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2000. – 176 с.
14. *Малышкин В.Э., Корнев В.Д.* Параллельное программирование мультикомпьютеров. – Новосибирск: Изд-во НГТУ, 2006. – 296 с.
15. http:/www2.sscc.ru/Publikacii/Primery_Prll
16. *Фадеев Д.К., Фадеева В.Н.* Вычислительные методы линейной алгебры. – М.; Л.: Физматгиз, 1969. – 738 с.
17. *Шпаковский Г.И., Серикова Н.В.* Программирование для многопроцессорных систем в стандарте MPI. – Минск: Изд-во БГУ, 2002. – 323 с.
18. *Самарский А.А., Николаев Е.С.* Методы решения сеточных уравнений. – М.: Наука, 1978. – 592 с.

19. *Хокни Р., Джесссхоуп К.* Параллельные ЭВМ. – М.: Радио и связь, 1986. – 392 с.

20. *Ильин В.П.* Методы неполной факторизации для решения алгебраических систем. – М.: Наука, 1995. – 287 с.

21. *Ортега Дж.* Введение в параллельные и векторные методы решения линейных систем. – М.: Мир, 1991. – 364 с.

22. *Самарский А.А.* Введение в численные методы. – М.: Наука, 1987. – 288 с.

23. *Самарский А.А.* Теория разностных схем. – М.: Наука, 1989. – 614 с.

24. *Корнеев В.В.* Параллельные вычислительные системы. – М.: Нолидж, 1999. – 320 с.

25. *Воеводин В.В., Воеводин Вл. В.* Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 609 с.

26. *Эндрюс Г.Р.* Основы многопоточного параллельного и распределенного программирования. – М.: Дом Вильямс, 2003. – 330 с.

27. *Барский А.Б.* Параллельные информационные технологии. – М.: БИНОМ, 2007. – 503 с.

28. *Богачев К.Ю.* Основы параллельного программирования. – М.: БИНОМ, 2003. – 342 с.

29. *Корнеев В.В.* Параллельное программирование в MPI. – Москва; Ижевск: Институт компьютерных исследований, 2003. – 303 с.

30. *Афанасьев К.Е., Домрачеев В.Г., Ретинская И.В., Скуратов А.К., Стуколов С.В.* Многопроцессорные системы: построение, развитие, обучение. – М.: КУДИЦ-ОБРАЗ, 2005. – 224 с.

31. *Немнюгин С.А.* Средства программирования для многопроцессорных вычислительных систем. – СПб.:СПбГУ, 2007. – 88 с.

32. *Гергель В.П.* Теория и практика параллельных вычислений. – М.: БИНОМ, 2007. – 424 с.

33. *Левин М.П.* Параллельное программирование с использованием OpenMP. – М.: БИНОМ, 2008. – 118 с.

34. *Воеводин В.В.* Численные методы, параллельные вычисления и информационные технологии. – М.: БИНОМ, 2008. – 320 с.

Учебное издание

Беликов Дмитрий Анатольевич
Говязов Иван Владимирович
Данилкин Евгений Александрович
Лаева Валентина Ивановна
Проханов Сергей Анатольевич
Старченко Александр Васильевич

Высокопроизводительные вычисления на кластерах

Учебное пособие

Редактор В.Г. Лихачева
Компьютерная верстка Е.А. Данилкин, С.А. Проханов

Лицензия ИД №04617 от 24.04.2001 г. Подписано в печать 20.08.2008 г.
Формат 60x84 $^{1}/_{16}$. Бумага офсетная №1. Печ. л. 12,3; усл. печ. л. 11,4; уч.-изд. л. 11,2.
Тираж 200 Заказ

ОАО «Издательство ТГУ», 634029, г. Томск, ул. Никитина, 4
ООО «Типография «Иван Федоров»», 634003, г. Томск, Октябрьский взвоз, 1