Г. И. Шпаковский,
А. Е. Верхотуров,
Н. В. Серикова

Руководство по работе на вычислительном кластере

Пособие для студентов
естественнонаучных
специальностей

Минск
БГУ
2004

В пособии рассматривается организация вычислительного кластера на базе локальной сети. Изложены способы инсталляции системного программного обеспечения, методы запуска параллельных задач, описаны средства оценки эффективности вычислений, представлено описание библиотеки для параллельных вычислений PETSc.

Предназначено студентам специальностей «Радиофизика» и «Физическая электроника». Издание может быть полезно студентам естественнонаучных специальностей и научным работникам для создания вычислительного кластера и изучения работы на нем.

ПРЕДИСЛОВИЕ

Под кластером понимают совокупность процессоров, объединенных компьютерной сетью и предназначенных для решения одной задачи, как правило, большой вычислительной сложности. Типы кластеров, их организация, возможности, математическое обеспечение, характеристики определенных кластеров представлены на сайтах [1–3]. Кластеры можно разделить на два класса:

1. Кластеры специальной разработки с быстродействием 10^{11}–10^{12} плавающих операций в секунду (терафлопсовый диапазон). К таким кластерам относится СКИФ (**С**упер**К**омпьютерная **И**нициатива **Ф**еникс), созданный по программе Союза Беларусь–Россия. Появление этого кластера и сам процесс разработки подготовили основу для развития параллельных вычислений в республике, вследствие чего возникла проблема широкой подготовки программистов для решения параллельных задач.
2. Кластеры, которые строятся на базе уже имеющихся локальных сетей из персональных компьютеров. Для создания кластеров в этом случае требуется только дополнительное программное обеспечение (ПО), поэтому их можно организовать в вузах и небольших организациях. Такие кластеры имеют относительно небольшое быстродействие, но их удобно использовать для обучения основам параллельного программирования и подготовки параллельных программ, которые затем могут выполняться на больших кластерах.

В Белгосуниверситете на факультете радиофизики и электроники (РФЭ) создан кластер, на котором несколько лет ведется обучение параллельным вычислениям. За это время коллективом преподавателей созданы методические руководства и издано пособие Г. И. Шпаковского, Н. В. Сериковой «Программирование для многопроцессорных систем в стандарте MPI» (Мн.: БГУ, 2002. 323 с.). Оно является пособием по написанию параллельных программ и широко используется в учебном процессе, обеспечивая также контролируемую самостоятельную работу (КСР).

Кластер – довольно сложный объект, и для работы на нем нужно иметь знания по сетям, устройству компьютеров, операционным системам, специальному программному обеспечению.

Настоящее учебное пособие предназначено для студентов старших курсов, магистрантов, аспирантов и научных работников. Оно позволяет научиться работать на кластере, как в процессе плановой учебной работы, так и в режиме КСР. Руководство предназначено в основном

для кластеров, работающих под управлением операционной системы (ОС) Windows NT, хотя в нем имеются сведения и для кластеров с ОС Linux. Базовым языком является С.

Пособие состоит из трех основных разделов. В первом разделе представлена организация сетей и кластеров на их основе, рассмотрены принципы параллельного программирования, состав системного пакета MPICH, принципы инсталляции и настройки системных средств кластера. Во втором разделе рассмотрены варианты запуска параллельных программ в одно- и многопроцессорном режимах, возможности отладки, профилирования программ, оценки их эффективности. В третьем разделе достаточно подробно представлено еще одно системное средство – библиотека параллельных программ PETSc, предназначенная для решения систем линейных алгебраических уравнений. Использование библиотеки позволяет достаточно быстро создавать сложные параллельные приложения.

Многие главы пособия заканчиваются контрольными вопросами, много информации размещено на сайте *http://www.cluster.bsu.by*, который создан на факультете РФЭ Белгосуниверситета.

Авторы выражают глубокую благодарность рецензентам: профессору М. К. Бузе и доценту Ю. И. Воротницкому, декану факультета радиофизики и электроники С. Г. Мулярчику, профессору А. Н. Курбацкому, преподавателям факультета: Г. К. Афанасьеву, Д. А. Стрикелеву, И. М. Шевкуну, А. С. Липницкому, В. А. Чудовскому за полезные замечания, способствовавшие улучшению содержания пособия. На создание пособия большое влияние оказало участие авторов в работе по проекту суперкомпьютера СКИФ.

Возможные предложения и замечания по содержанию пособия просим присылать по адресу:

Республика Беларусь
220050, Минск, проспект Франциска Скорины, 4
Белорусский государственный университет
Факультет радиофизики и электроники, кафедра информатики
E-mail: Serikova@bsu.by, Shpakovski@bsu.by, Verkhoturov@rambler.ru

I. ОРГАНИЗАЦИЯ КЛАСТЕРОВ

Глава 1. СЕТЕВЫЕ СРЕДСТВА КЛАСТЕРОВ

1.1. КЛАССЫ ПАРАЛЛЕЛЬНЫХ ЭВМ, КЛАСТЕРЫ

Классификация. Согласно классификации Флинна [4, 5] имеются два основных класса параллельных ЭВМ с крупнозернистым параллелизмом.

1. **SIMD** (**S**ingle **I**nstruction – **M**ultiple **D**ata) – одиночный поток команд и множественный поток данных. В таких ЭВМ выполняется единственная программа, но каждая команда обрабатывает массив данных. Это соответствует векторной форме параллелизма.
2. **MIMD** (**M**ultiple **I**nstruction – **M**ultiple **D**ata) – множественный поток команд и множественный поток данных. В таких ЭВМ одновременно и независимо друг от друга выполняется несколько программных ветвей, обменивающихся данными. Такие системы обычно называют многопроцессорными. Кластеры относятся к классу MIMD.

Все параллельные ЭВМ предназначены для уменьшения времени решения задач с большим и сверхбольшим временем счета. Основной характеристикой параллельных ЭВМ является ускорение R, определяемое выражением

$$R = T_1 / T_n,$$

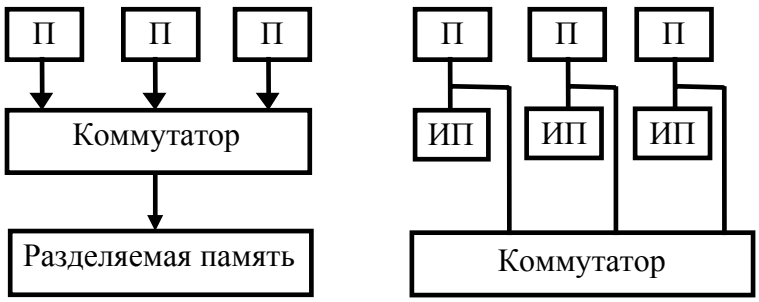
где T_1 – время выполнения задачи на однопроцессорной ЭВМ; T_n – время выполнения той же задачи на n-процессорной ЭВМ.

В основе MIMD-ЭВМ лежит традиционная последовательная организация программы, расширенная добавлением специальных способов для указания независимых фрагментов, которые можно выполнять параллельно. Параллельно-последовательная программа привычна для пользователя и позволяет относительно просто собирать параллельную программу из обычных последовательных программ.

MIMD-ЭВМ имеют две разновидности: ЭВМ с разделяемой (общей) и распределенной (индивидуальной) памятью. Структура этих ЭВМ представлена на рис. 1.1.

Главное различие между MIMD-ЭВМ с общей и индивидуальной памятью состоит в характере адресной системы.

В машинах с общей памятью адресное пространство всех процессоров является единым, следовательно, если в программах нескольких



а *б*

Рис.1.1. Структура ЭВМ с разделяемой (*а*) и индивидуальной (*б*) памятью. *П* – процессор; *ИП* – индивидуальная память.

процессоров встречается одна и та же переменная *X*, то все процессоры будут обращаться в одну физическую ячейку общей памяти. Это вызывает как положительные (не нужно физически перемещать данные), так и отрицательные последствия, поскольку одновременное обращение нескольких процессоров к общим данным может привести к получению неверных результатов. Чтобы исключить такие ситуации, необходимо ввести систему синхронизации параллельных процессов (например, семафоры), что усложняет механизмы операционной системы [6].

Так как при выполнении каждой команды процессорам необходимо обращаться в общую память, требования к пропускной способности этой памяти чрезвычайно высоки, что ограничивает число процессоров в системах величиной 10–20.

В системах с индивидуальной памятью (с обменом сообщениями) каждый процессор имеет независимое адресное пространство, и наличие одной и той же переменной *X* в программах разных процессоров приводит к обращению в физически разные ячейки памяти. Это вызывает физическое перемещение данных между взаимодействующими программами в разных процессорах, однако поскольку основная часть обращений в каждом процессоре осуществляется в собственную память, то требования к коммутатору ослабляются, и число процессоров в системах с распределенной памятью и скоростным коммутатором может достигать нескольких сотен и даже тысяч.

Существующие параллельные вычислительные средства класса MIMD образуют три технических подкласса:

- симметричные мультипроцессоры;
- системы с массовым параллелизмом;
- кластеры.

Симметричные мультипроцессоры (SMP – Symmetric **M**ulti **P**rocessors**)** используют принцип разделяемой памяти. В этом случае система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой ячейке памяти с одинаковой скоростью. Процессоры подключены к памяти с помощью общей шины или коммутатора. Аппаратно поддерживается когерентность кэшей. Вся система работает под управлением единой ОС.

Системы с массовым параллелизмом (MPP – Massively **P**arallel **P**rocessing**)** содержат множество процессоров с индивидуальной памятью, которые связаны через некоторую коммуникационную среду. Как правило, системы MPP благодаря специализированной высокоскоростной системе обмена обеспечивают наивысшее быстродействие (и наивысшую стоимость) и находятся во главе списка 500 самых быстрых ЭВМ [7].

Кластерные системы – более дешевый вариант MPP-систем, поскольку они также используют принцип передачи сообщений, но строятся из компонентов высокой степени готовности [8, 9]. Базовым элементом кластера является локальная сеть. Оказалось, что на многих классах задач и при достаточном числе узлов такие системы дают производительность, сравнимую с суперкомпьютерной.

Первым кластером на рабочих станциях был **Beowulf** [1, 2]. Проект Beowulf начался в 1994 г. сборкой в научно-космическом центре NASA 16-процессорного кластера на Ethernet-кабеле. С тех пор кластеры на рабочих станциях обычно называют Beowulf-кластерами. Любой Beowulf-кластер состоит из машин (узлов) и объединяющей их сети (коммутатора). Кроме ОС, необходимо установить и настроить сетевые драйверы, компиляторы, ПО поддержки параллельного программирования и распределения вычислительной нагрузки. В качестве узлов обычно используются однопроцессорные ПЭВМ с быстродействием 1 ГГц и выше или SMP-серверы с небольшим числом процессоров (обычно 2–4).

Для получения хорошей производительности межпроцессорных обменов используют полнодуплексную сеть **Fast Ethernet** с пропускной способностью 100 Мбит/с. При этом для уменьшения числа кол-

лизий устанавливают несколько «параллельных» сегментов Ethernet или соединяют узлы кластера через коммутатор (switch).

В качестве операционных систем обычно используют **Linux** или **Windows NT** и ее варианты, а в качестве языков программирования – C, C++ и старшие версии языка Fortran.

Наиболее распространенным интерфейсом параллельного программирования в модели передачи сообщений является **MPI** (**M**essage **P**assing **I**nterface) [1, 2, 8]. Рекомендуемой бесплатной реализацией MPI является пакет **MPICH** [1], разработанный в Аргоннской национальной лаборатории США.

Во многих организациях имеются локальные сети компьютеров с соответствующим программным обеспечением. Если такую сеть снабдить бесплатной реализацией MPI, т. е. пакетом MPICH, то без дополнительных затрат получается Beowulf-кластер, сравнимый по мощности с супер-ЭВМ. Это является причиной широкого распространения таких кластеров.

1.2. КОМПЬЮТЕРНЫЕ СЕТИ

Основой любого кластера является компьютерная сеть. Компьютерная сеть – сложный комплекс взаимосвязанных и согласованно функционирующих программных и аппаратных компонентов [10].

Элементы сети:

- компьютеры;
- коммуникационное оборудование;
- операционные системы;
- сетевые приложения.

Хронологически первыми появились глобальные сети. Они соединяют компьютеры, рассредоточенные на расстоянии сотен и тысяч километров. Эти сети очень многое унаследовали от телефонных сетей. В основном они предназначены для передачи данных.

Локальные сети сосредоточены на территории не более 1–2 км, построены с использованием высококачественных линий связи, которые позволяют достигать высоких скоростей обмена данными более 100 Мбит/с. Именно эти сети используются для построения кластеров.

Адресация в сетях. Множество адресов, допустимых в рамках некоторой схемы адресации, называется *адресным пространством*. Адресное пространство может иметь плоскую (линейную) или иерархическую организацию.

В первом случае множество адресов не структурировано, все адреса равноправны. Во втором случае оно организовано в виде вложенных друг в друга подгрупп, которые, последовательно сужая адресную область подобно почтовому адресу (страна, город, улица, дом, квартира), в конце концов определяют отдельный сетевой интерфейс.

Примером плоского числового адреса является **MAC** (**M**edia **A**ccess **C**ontrol) адрес, предназначенный для однозначной идентификации сетевых интерфейсов в локальных сетях. Такой адрес обычно используется только аппаратурой, поэтому его стараются сделать по возможности компактным и записывают в виде 6-байтового числа, например 0081005e24a8. При задании адресов не нужно выполнять ручную работу, так как адреса встраиваются в аппаратуру компанией-изготовителем, поэтому их называют аппаратными. Адреса MAC между изготовителями распределяются централизованно комитетом IEEE.

Типичным представителем иерархических числовых адресов являются 4-байтовые сетевые **IP** (**I**nternet **P**rotocol) адреса, используемые для адресации узлов в глобальных сетях. В них поддерживается двухуровневая иерархия, адрес делится на старшую (номер сети) и младшую (номер узла в сети) часть. Такое деление позволяет передавать сообщение между сетями только на основании номера сети, а номер узла используется после доставки сообщения в нужную сеть.

В современных узлах для адресации применяют обе системы: передача между сетями осуществляется с помощью адресов IP, а внутри сети – с помощью адресов MAC. Кроме того, для удобства используются символические имена.

В связи с множественностью систем адресации возникла проблема установления соответствия между адресами различных типов, которая решается двояко. При централизованном решении в сети устанавливается один или несколько компьютеров, в которых хранятся таблицы соответствия адресов, а все остальные обращаются к ним для получения необходимой информации. В распределенном случае каждый компьютер выявляет соответствие сетевого и аппаратного адреса путем опроса соседей, каждый из которых знает оба своих адреса.

Сетевые протоколы. Формализованные правила, определяющие последовательность и формат сообщений, которыми обмениваются сетевые компоненты одного уровня различных узлов, называют *протоколом*. Для обмена используется набор протоколов различного уровня. Эти протоколы должны согласовывать величину, форму элек-

трических сигналов, способ определения размера сообщения, методы контроля достоверности и др.

Модули, реализующие протоколы соседних уровней и находящиеся в одном узле, также взаимодействуют друг с другом в соответствии с четко определенными правилами и с помощью стандартизованных форматов сообщений. Эти правила принято называть интерфейсом. Следовательно, протоколы определяют правила взаимодействия модулей одного уровня в разных узлах, а интерфейсы – правила взаимодействия модулей соседних уровней в одном узле.

Иерархически организованный набор протоколов, достаточный для организации взаимодействия узлов сети, называется *стеком коммуникационных протоколов*.

Модель OSI (**O**pen **S**ystem **I**nterconnection). Основным стандартом набора коммуникационных протоколов является модель OSI, разработанная в начале 80-х годов [10]. В модели OSI (рис.1.2) средства взаимодействия делятся на семь уровней: прикладной, представительный, сеансовый, транспортный, сетевой, канальный и физический.

Рассмотрим работу модели OSI при выполнении обмена между двумя узлами. Пусть приложение обращается с запросом к прикладному уровню, например к файловой службе. На основании этого запроса программы прикладного уровня формируют сообщение стандартного формата. Обычное сообщение состоит из заголовка и поля данных. Заголовок содержит служебную информацию, которую необходимо передать через сеть прикладному уровню машины-адресата, чтобы сообщить ему, какую работу надо выполнить. В нашем случае заголовок должен содержать информацию о месте нахождения файла и типе операции, которую необходимо над ним выполнить.

После формирования сообщения прикладной уровень направляет его вниз по стеку представительному уровню. Протокол представительного уровня на основании информации, полученной из заголовка прикладного уровня, выполняет требуемые действия и добавляет к сообщению собственную служебную информацию – заголовок представительного уровня, в котором содержатся указания для протокола представительного уровня машины-адресата.

Полученное в результате сообщение передается вниз сеансовому уровню, который добавляет свой заголовок и т. д. Наконец, сообщение достигает нижнего физического уровня, который собственно и передает его по линии связи машине-адресату.

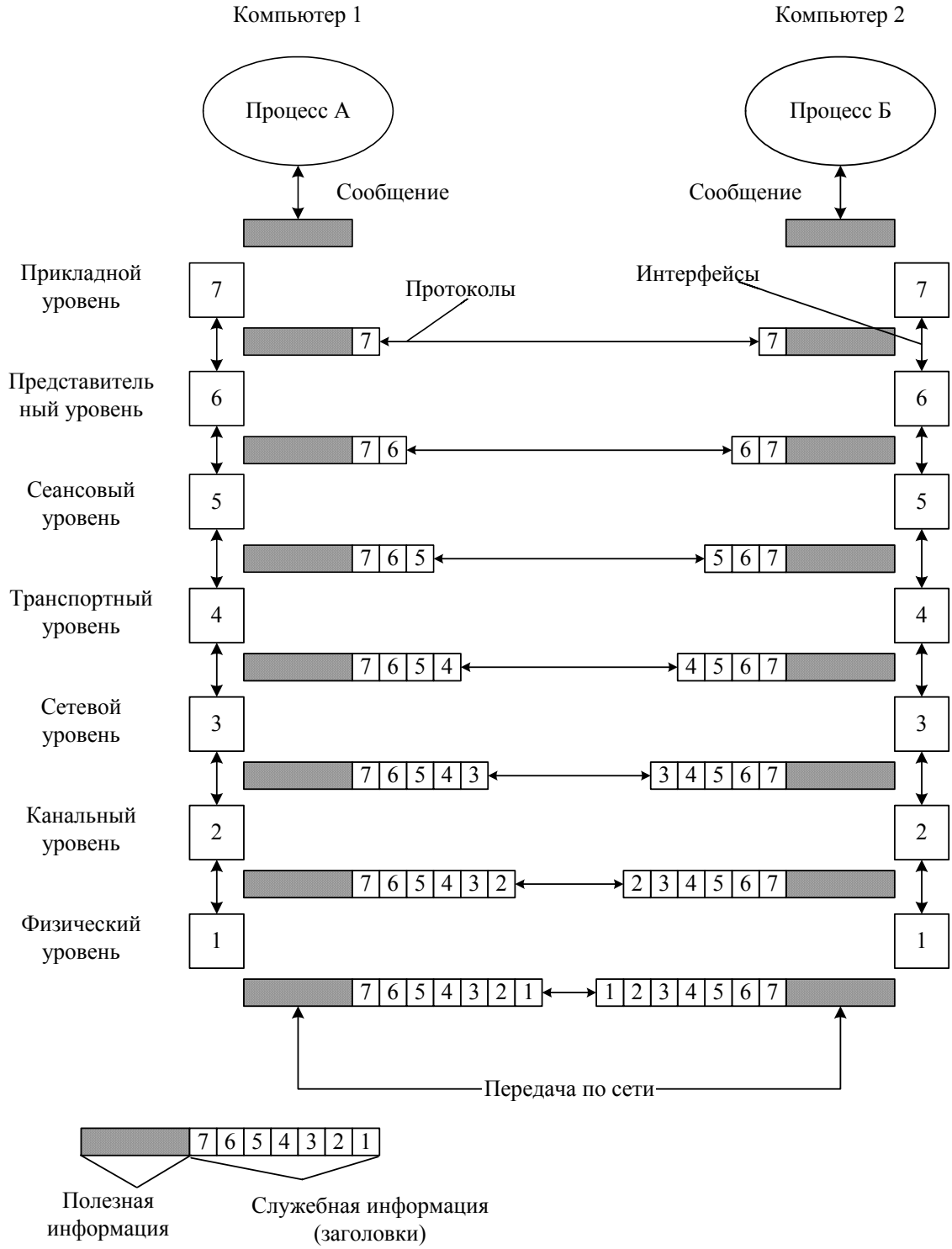


Рис. 1.2. Модель взаимодействия открытых систем OSI.

К этому моменту сообщение обрастает заголовками всех уровней (рис. 1.3). Когда сообщение по сети поступает на машину-адресат, оно принимается ее физическим уровнем и последовательно перемещается вверх с уровня на уровень. Каждый уровень анализирует и обрабатывает заголовок своего уровня, выполняя соответствующие данному уровню функции, а затем удаляет этот заголовок и передает сообщение вышележащему уровню.

В модели OSI уровни выполняют различные функции.

- Физический уровень – самый нижний. На этом уровне определяются характеристики электрических сигналов, передающих дискретную информацию, например крутизна фронтов импульсов, уровни напряжения или тока передаваемых сигналов. Кроме того, здесь стандартизуются типы разъемов и назначение каждого контакта.
- Канальный уровень по-разному определяется в технологиях локальных и глобальных сетей. В глобальных сетях, для которых характерна произвольная топология, на канальном уровне решаются проблемы надежной связи двух соседних узлов. В локальных сетях с типовой топологией (например, звезда, кольцо, общая шина) на этом уровне решается задача обеспечить надежную связь любой пары узлов. Канальный уровень отвечает за формирование кадров, физическую адресацию, разделение передающей среды, контроль ошибок.
- Сетевой уровень служит для образования единой транспортной системы, объединяющей несколько сетей. Он манипулирует дейтаграммами.
- Транспортный уровень обеспечивает прикладному и сеансовому уровням передачу данных с той степенью надежности, которая им требуется.
- Сеансовый уровень осуществляет управление взаимодействием: фиксирует, какая из сторон является активной в настоящий момент, предоставляет средства синхронизации.
- Представительный уровень имеет дело с формой представления передаваемой информации, не меняя при этом ее содержания.

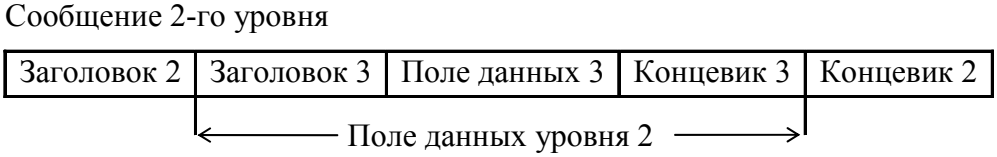


Рис.1.3. Взаимосвязь по информации между смежными уровнями

• Прикладной уровень – это набор разнообразных протоколов, с помощью которых пользователи сети получают доступ к разделяемым ресурсам.

Единица данных, которой оперируют три верхних уровня, называется *сообщением* (message).

Ethernet – самая распространенная технология на физическом и канальном уровнях, она является основой локальных сетей. Технологии Ethernet, Fast Ethernet, Gigabit Ethernet обеспечивают скорость передачи данных соответственно 10, 100 и 1000 Мбит/с. Все технологии используют один метод доступа – CSMA/CD (**C**arrier **S**ense **M**ultiply **A**ccess with **C**ollision **D**etection), одинаковые форматы кадров, работают в полу- и полнодуплексном режимах. Метод предназначен для среды, разделяемой всеми абонентами сети.

На рис. 1.4 представлен метод доступа CSMA/CD. Чтобы получить возможность передавать кадр, абонент должен убедиться, что среда свободна. Это достигается прослушиванием несущей частоты сигнала. Отсутствие несущей частоты является признаком свободности среды.

На рис. 1.4 узел *1* обнаружил, что среда сети свободна, и начал передавать свой кадр. В классической сети Ethernet на коаксиальном кабеле сигналы передатчика узла *1* распространяются в обе стороны, поэтому все узлы сети их получают. Все станции, подключенные к кабелю, могут распознать факт передачи кадра, и та станция, которая узнает свой адрес в заголовке передаваемого кадра, записывает его содержимое в свой внутренний буфер, обрабатывает полученные данные, передает их вверх по своему стеку, а затем посылает по кабелю

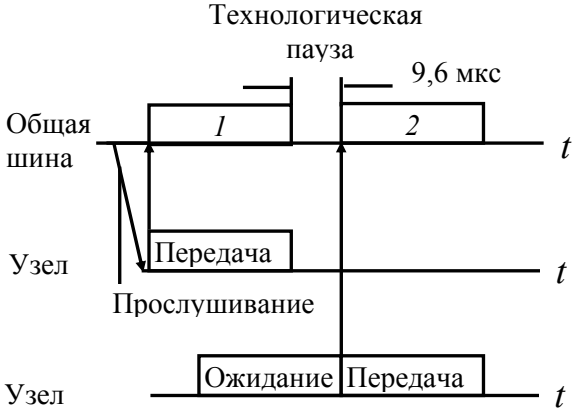


Рис. 1.4. Метод случайного доступа CSMA/CD

кадр-ответ. Адрес станции-источника содержится в исходном кадре, поэтому станция-получатель знает, кому послать ответ.

Узел *2* во время передачи кадра узлом *1* также пытался начать передачу своего кадра, однако обнаружил, что среда занята – на ней присутствует несущая частота, поэтому узел *2* вынужден ждать, пока узел *1* не прекратит передачу кадра.

После передачи кадра все узлы обязаны выдержать технологическую паузу в 9,6 мкс. Этот межкадровый интервал нужен для приведения сетевых адаптеров в исходное состояние, а также для предотвращения монопольного захвата среды одной станцией. После окончания технологической паузы узлы имеют право начать передачу своего кадра, так как среда свободна. В приведенном примере узел *2* дождался передачи кадра узлом *1*, сделал паузу в 9,6 мкс и начал передачу своего кадра.

В разделяемой среде возможны ситуации, когда две станции пытаются одновременно передать кадр данных, при этом происходит искажение информации. Такая ситуация называется *коллизией*. В Ethernet предусмотрен специальный механизм для разрешения коллизий [10]. В большинстве случаев рабочие станции в сети на витой паре объединяются с помощью концентраторов (concentrator, hub) или коммутаторов (commutator, switch). На рис. 1.5 показано соединение с помощью концентратора.

Концентратор осуществляет функцию общей среды, т. е. является логической общей шиной. Каждый конечный узел соединяется с кон-

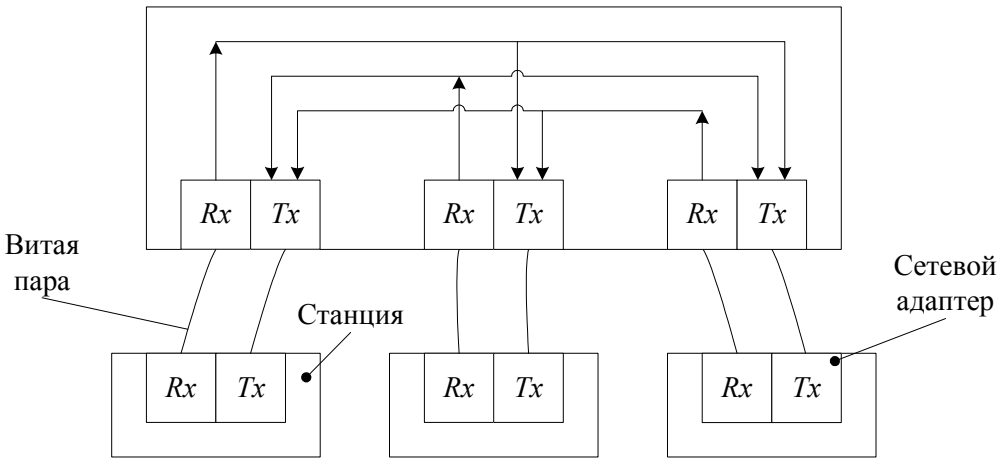


Рис. 1.5. Сеть стандарта 100 Base-T4: T_x – передатчик; R_x - приемник

центратором с помощью двух витых пар. Одна витая пара требуется для передачи данных от станции к порту концентратора (выход T_x сетевого адаптера), а другая – для передачи данных от порта концентратора (вход R_x сетевого адаптера). Концентратор принимает сигналы от одного из конечных узлов и синхронно передает их на все свои остальные порты, кроме того, с которого поступили сигналы.

Основной недостаток сетей Ethernet состоит в том, что в них в единицу времени может передаваться только один пакет данных. Обстановка усугубляется задержками доступа из-за коллизий. На рис. 1.6 представлен график пропускной способности сети в зависимости от коэффициента использования сети ρ (отношение задаваемой нагрузки к максимально возможной).

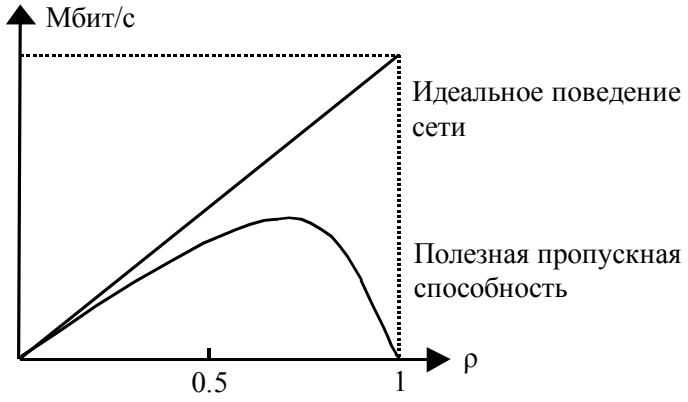


Рис. 1.6. Зависимость полезной пропускной способности сети Ethernet от коэффициента использования

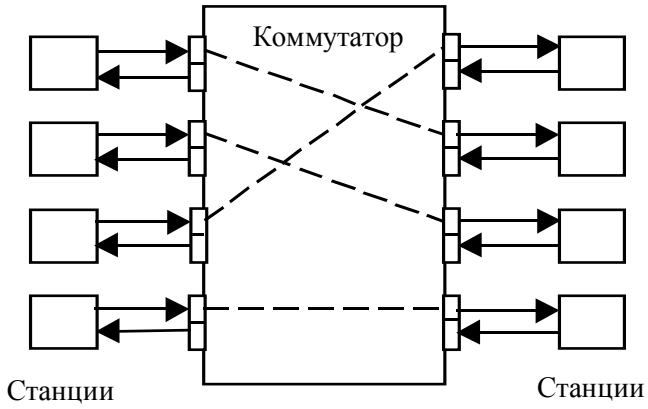


Рис. 1.7. Параллельная передача кадров коммутатором

Следовательно, сеть типа Ethernet не следует загружать более чем на 30–40% ее максимальной пропускной способности.

Пропускная способность Ethernet сетей значительно повышается при использовании коммутатора. Существует много типов коммутаторов [5]. На рис. 1.7 представлена схема коммутатора типа «координатный переключатель». В таком коммутаторе каждый порт связан электрически со всеми другими портами. Очевидно, что в таком коммутаторе могут одновременно выполняться n пар обменов, где n – число портов. На рис. 1.7 представлен частный случай логического замыкания портов.

1.3. СЕТЕВЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Термин «операционная система» обозначает программное обеспечение, которое управляет компьютером. Термин «сетевая операционная система» обозначает программное обеспечение, которое управляет работой компьютеров в сети. Сетевая операционная система используется в кластере для выполнения следующих функций:

- обеспечения запуска процессов;
- осуществления передачи данных между процессами;
- доступа к удаленной файловой системе и устройствам вывода.

Эти функции обеспечиваются как реализациями MPI (будут рассмотрены в гл. 3), так и протоколами удаленного доступа, которые заложены в сетевые операционные системы. Удаленный доступ в этих системах реализован по-разному, причем он более развит в реализациях ОС Unix. Это объясняется несколькими причинами:

- ОС Unix появилась на свет гораздо раньше, чем Windows, поэтому в ней накоплен значительно больший объем полезных средств различного назначения и они хорошо отработаны;
- в ОС Unix для удаленного доступа с самого начала используется протокол **TCP/IP** (**T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol), получивший преобладающее распространение как в глобальных (Интернет), так и в локальных сетях. Стек TCP/IP характерен своими сетевым и транспортным уровнями и предназначен для маршрутизации и достоверной передачи сообщений в глобальных сетях, в частности в Интернет. На сегодня это самый отработанный и популярный стек протоколов;
- наконец, появилась бесплатно распространяемая реализация Unix, которая называется Linux.

В настоящее время в сетях используется большое количество стеков коммуникационных протоколов. На рис. 1.8 представлены некоторые из них.

Модель OSI	IBM/Microsoft		TCP/IP	Novell
1	2	3	4	5
Прикладной	SMB, MPICH	MPICH	Telnet, FTP, MPICH и др.	NSP, SAP
Представительный				
Сеансовый	NetBIOS NBF	NBT	TCP	
Транспортный		TCP		SPX
Сетевой		IP	IP, RIP, OSPF	IPX, RIP, NLSP
Канальный	Протоколы Ethernet, Fast Ethernet, Token Ring, ATM, X25 и др.			
Физический	Коаксиал, витая пара, оптоволокно			

Рис.1.8. Соответствие популярных стеков протоколов модели OSI

Для Windows NT возможны следующие варианты организации обмена в кластере.

1. С использованием «родного» стека протоколов: NetBIOS – NBF (столбец 1). Протокол NBF выполняет функции протокола сетевого и транспортного уровней. При сетевом вводе-выводе верхние уровни (NetBIOS, CMB) передают данные прямо в NBF для обработки. NBF инкапсулирует данные в кадры, находит устройство или устройства, с которыми нужно установить связь, и передает данные в сетевую интерфейсную плату для последующей доставки. Этот вариант изначально создавался как высокоэффективный протокол для малых сетей, поэтому он отличается высоким быстродействием. К сожалению, этот стек не обеспечивает маршрутизацию сообщений.
2. Чтобы использовать совместно с протоколом TCP/IP сетевые средства ОС Windows NT, необходимо иметь механизм преобразования имен NetBIOS в IP адреса и обратно. Этот механизм называется NetBIOS-над-TCP/IP (NetBIOS-over-TCP/IP, NBT). Это и показано на рис. 1.8 (столбец 2).

3. В последних вариантах реализации для Windows NT используется прямое преобразование формата команд MPI в форматы протокола TCP/IP (столбец 3), как это делается и в ОС Linux (столбец 4).

1.4. ХАРАКТЕРИСТИКИ НЕКОТОРЫХ КОМПЬЮТЕРНЫХ СЕТЕЙ

Коммуникационная производительность кластерной системы при передаче сообщения от узла A к узлу B определяется выражением

$$T(L) = s + L/R,$$

где s – латентность; L – длина сообщения; а R – пропускная способность канала связи.

Латентность (задержка) – это промежуток времени между запуском операции обмена в программе пользователя и началом реальной передачи данных в коммуникационной сети. Латентность содержит две составляющие:

- время выполнения всех операций в MPICH, связанных с преобразованием формата функции обмена MPI в формат пакета для TCP/IP;
- время прохождения уровней TCP/IP до начала передачи данных.

Принято считать, что время латентности делится примерно поровну между MPICH и TCP/IP [9]. В табл. 1.1 представлены характеристики некоторых компьютерных сетей при работе с пакетом MPICH.

Хорошими характеристиками обладает получившая большое распространение коммуникационная технология **SCI** (**S**calable **C**oherent **I**nterface) [11]. Основа технологии SCI – кольца, состоящие из быстрых однонаправленных соединений с пиковой пропускной способностью на аппаратном уровне 400 Мбайт/с. Реальная пропускная способность на уровне MPI-приложений с использованием 32-разрядной шины PCI с частотой 33 МГц достигает 80 Мбайт/с, латентность – 5,6 мкс. Основной поставщик промышленных SCI-компонентов на современном рынке – норвежская компания Dolphin Interconnect Solutions, которая вместе с компанией Scali Computer предлагает интегрированное кластерное решение Wulfkit. В состав последнего, в частности, входит пакет ScaMPI – оптимизированная под SCI реализация интерфейса MPI.

При построении кластеров на основе рабочих станций или персональных компьютеров всегда возникает вопрос, какая операционная система более пригодна для реализации обменов – Linux или сетевой вариант Windows.

Таблица 1.1

Характеристики некоторых коммуникационных технологий

Характеристика	SCI	Myrinet	cLAN	Fast Ethernet
Латентность в мкс	5,6	17	30	170
Пропускная способность (MPI) в Мбайт/с	80	40	100	10
Пропускная способность (аппаратная) в Мбайт/с	400	160	150	12,5
Реализация MPI	ScaMPI	HPVMи др.	MPI /Pro	MPICH

Такие исследования проводились, и некоторые результаты работы [12] представлены на рис. 1.9.

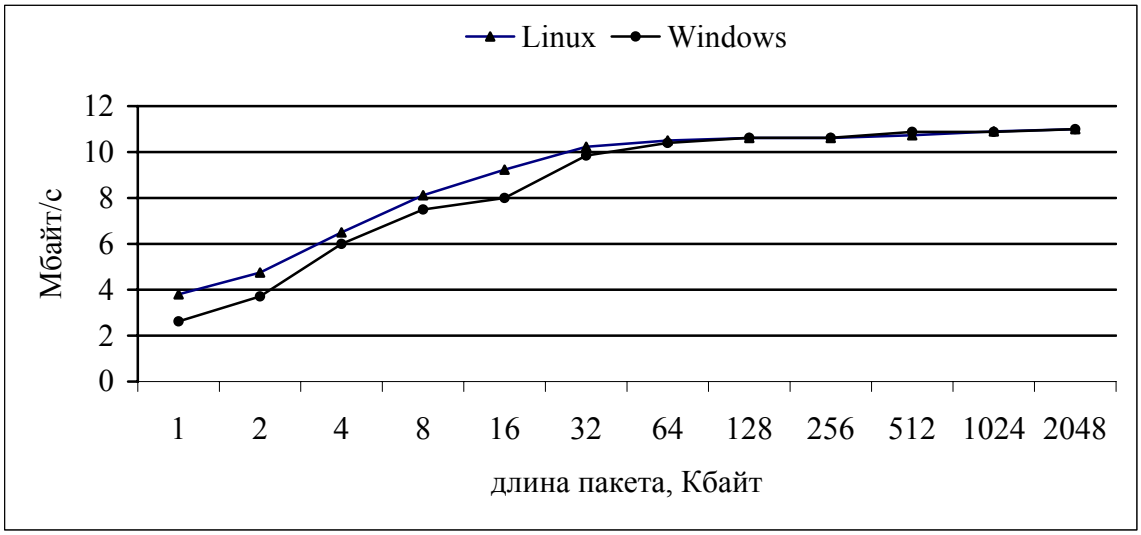


Рис. 1.9. Скорость передачи данных между узлами кластера для операционных систем Linux и Windows

Сравнение результатов показывает, что в случае обменов короткими сообщениями MPICH для ОС Linux обеспечивает более высокую скорость передачи данных и меньшую латентность, чем MPICH для ОС Windows. Однако при увеличении размера передаваемых пакетов преимущество реализации MPICH для Linux не выражено, и скорости передачи данных для обеих реализаций становятся практически одинаковыми.

1.5. КЛАСТЕР СКИФ

В соответствии с работой [9] кластеры можно разделить на две категории.

- Кластеры типа Beowulf (или кластеры невыделенных рабочих станций). Рабочие станции в таком кластере могут использоваться как для параллельных вычислений под MPICH, так и для обычной работы в локальной сети или изолированно.
- Кластеры, выполненные в специализированном компактном конструктиве (или кластеры выделенных рабочих станций). Малые физические размеры кластера позволяют применить коммуникационные сети с очень высоким быстродействием.

Эти факторы позволяют получить от специализированных кластеров (по отношению к Beowulf-кластерам) быстродействие в терафлопсовом диапазоне. Примером такого кластера является кластер СКИФ, разработанный по программе Союзного государства Россия – Беларусь [2]. Схема кластера представлена на рис. 1.10.

СКИФ имеет двухуровневую архитектуру.

- Кластерный уровень (КУ), который состоит из множества SMP процессоров, соединенных системной коммуникационной сетью. Вспомогательная сеть обеспечивает доступ к кластеру от множества удаленных пользователей.
- Потоковый уровень, или уровень однородной вычислительной среды (ОВС). После заданной настройки ОВС работает в чисто аппаратном режиме, и при поступлении на вход непрерывного потока данных (например, от радиолокационных антенн) может обеспечивать очень высокое быстродействие.

Объединение в единой установке двух разных уровней основано на следующей идее – в общем случае каждое приложение может быть разбито на фрагменты:

- со сложной логикой вычислений, с крупноблочным параллелизмом могут быть эффективно реализованы на кластерном уровне;
- с простой логикой вычислений, с конвейерным или мелкозернистым явным параллелизмом, с большими потоками информации, требующими обработки в реальном режиме времени, могут быть эффективно реализованы на ОВС.

После анализа прикладной задачи в рамках идеи составного суперкомпьютера можно подобрать оптимальную пропорцию аппаратных средств для решения конкретной задачи, оптимальное число вычислительных модулей кластерного уровня и блоков ОВС.

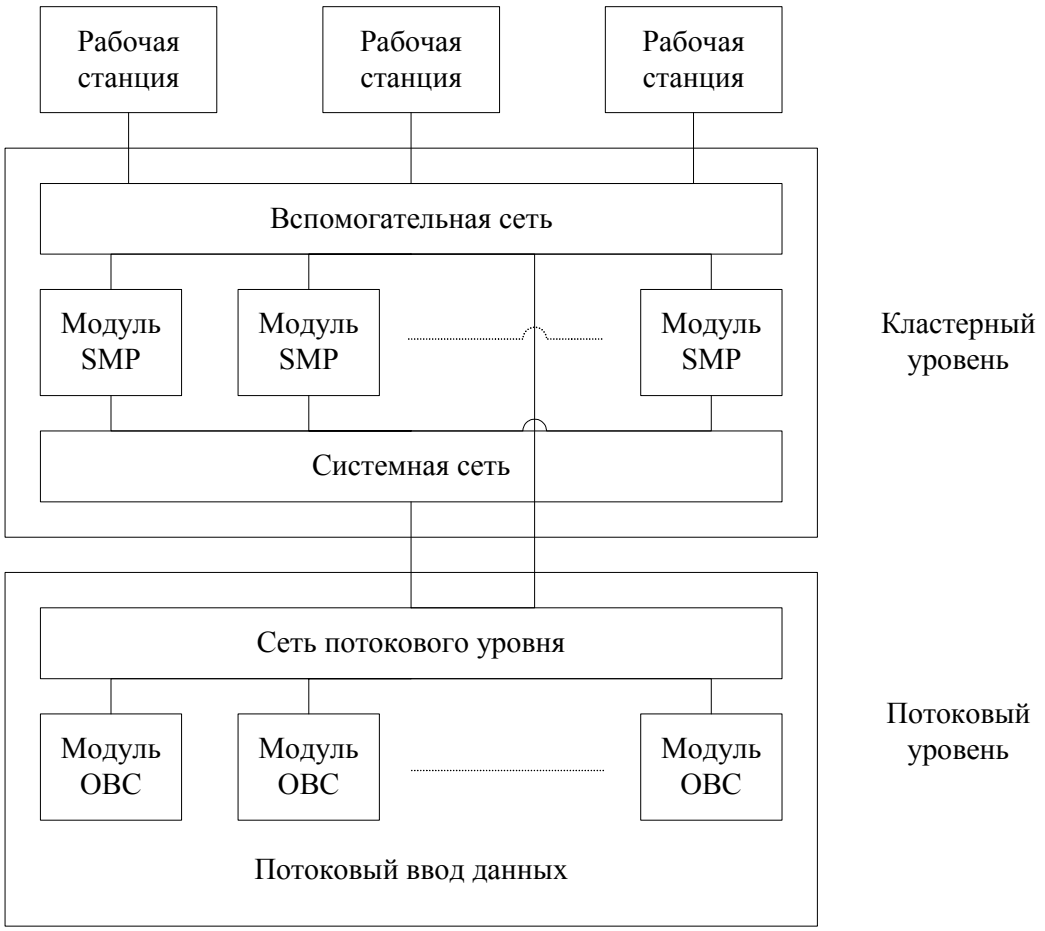


Рис. 1.10. Схема кластера СКИФ

Архитектура суперкомпьютеров, разрабатываемых по программе «СКИФ», является открытой и масштабируемой. В ней не накладывается никаких жестких ограничений на аппаратную платформу вычислительных модулей кластерного уровня, на аппаратуру и топологию системной сети, на конфигурацию и диапазон производительности разрабатываемых суперкомпьютеров.

1.6. МЕТАКОМПЬЮТИНГ

Кластер – параллельный компьютер, все процессоры которого действуют как единое целое для решения одной задачи. Совокупность таких компьютеров составляет метакомпьютер [4]. Примером метакомпьютеров являются глобальные сети, в том числе и Интернет, обладающие неограниченной мощностью. Однако чтобы преобразовать Интернет в метакомпьютер, необходимо разработать большой объем системного математического обеспечения. В комплексе должны рас-

сматриваться такие вопросы, как средства и модели программирования, распределение и диспетчирование заданий, технология организации доступа к метакомпьютеру, интерфейс с пользователями, безопасность, надежность, политика администрирования, средства доступа и технологии распределенного хранения данных, мониторинг состояния различных подсистем метакомпьютера и др. Сложно заставить десятки миллионов различных электронных устройств, составляющих метакомпьютер, работать согласованно над заданиями десятков тысяч пользователей в течение продолжительного времени.

Одним из проектов организации метакомпьютинга является проект Globus, который изначально зародился в Аргоннской национальной лаборатории и сейчас получил широкое распространение во всем мире. Цель проекта – создание средств для организации глобальной информационно-вычислительной среды. В рамках проекта разработаны программные средства и системы, в частности единообразный интерфейс к различным локальным системам распределения нагрузки, системы аутентификации, коммуникационная библиотека Nexus, средства контроля и мониторинга и др. Разработанные средства распространяются свободно в виде пакета Globus Toolkit вместе с исходными текстами. В настоящее время Globus взят за основу в множестве других масштабных проектов, таких как National Technology Grid, Information Power и Grid European DataGrid. Дополнительную информацию можно найти на сайте *http://www.globus.org*.

В настоящее время выполняется несколько задач метакомпьютерного уровня [4]:

- поиск внеземных цивилизаций с помощью распределенной обработки данных, поступающей с радиотелескопа. К этой работе может подключиться каждый желающий;
- решение задач криптоанализа по проверке стойкости криптосистем;
- характерной задачей для метакомпьютинга является создание информационной инфраструктуры для поддержки экспериментов в физике высоких энергий.

В 2006–2007 годах в Европейской организации ядерных исследований (CERN) планируется ввести в строй мощный ускоритель – Большой Адронный Коллайдер. Четыре физических установки данного проекта в течение 15–20 лет будут ежегодно собирать данные объемом порядка нескольких Пбайт (10^{15} байт). Во время эксперимента данные будут поступать со скоростью от 100 Мбайт/с до 1 Гбайт/с.

Оптимальная схема хранения и последующая эффективная обработка данных, собранных тысячами участников эксперимента из разных стран мира, является исключительно сложной задачей.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите и опишите классы параллельных ЭВМ по Флинну.
2. Что такое многопроцессорные ЭВМ с общей памятью?
3. Что такое многопроцессорные ЭВМ с индивидуальной памятью?
4. Что вызывает некорректность вычислений в ЭВМ с общей памятью?
5. Каковы достоинства и недостатки ЭВМ с передачей сообщений?
6. Что такое кластер?
7. В чем состоит принципиальное отличие кластеров от систем MPP?
8. Назовите и опишите два основных типа адресации в компьютерных сетях.
9. Что такое сетевой протокол?
10. Назовите и опишите уровни протоколов модели OSI.
11. Что такое межуровневый интерфейс?
12. Опишите структуру сообщения некоторого уровня.
13. Какие программные и аппаратные средства соответствуют канальному и физическому уровням?
14. Что такое технология Ethernet и метод доступа CSMA/CD?
15. Опишите структуру концентратора в сети Ethernet.
16. Какова допустимая загрузка сети Ethernet?
17. Чем отличается коммутатор от концентратора?
18. Что такое сетевая операционная система для кластера?
19. Чем отличаются сетевые операционные системы Windows NT и Linux?
20. Опишите возможные уровни протоколов для Windows NT и TCP/IP
21. Что такое латентность, как она определяется?
22. От чего зависит скорость обменов в кластере?
23. Каковы характеристики основных компьютерных сетей?
24. Как влияет на скорость обмена тип операционной системы?
25. Кем и с какой целью финансировалась разработка проекта СКИФ?
26. Опишите составные части кластера СКИФ.
27. Чем кластер СКИФ отличается от кластеров рабочих станций?
28. Что такое метакомпьютер?
29. Для каких целей используется метакомпьютинг?
30. Что такое проект Globus, кем он реализован?

Глава 2. СИСТЕМНЫЕ ПРОГРАММНЫЕ СРЕДСТВА КЛАСТЕРОВ

2.1. СИСТЕМЫ ПРОГРАММИРОВАНИЯ ДЛЯ КЛАСТЕРОВ

Основными средствами программирования для многопроцессорных систем являются две библиотеки, оформленные как стандар-

ты: библиотека OpenMP для систем с общей памятью (для SMP-систем) и библиотека MPI для систем с индивидуальной памятью.

Определенное распространение получила и библиотека PVM (**P**arallel **V**irtual **M**achine), которая также предназначена для параллельных вычислений на машинах с индивидуальной памятью [13]. Она близка по значению и набору функций к библиотеке MPI, однако спецификация PVM не регламентируется каким-либо из общепринятых стандартов, поэтому PVM не используется в крупных проектах. Документация с описанием по библиотеки PVM доступна в по адресу: *http://www.csm. ornl.gov/pvm/pvm_home.html*.

2.1.1. Стандарт OpenMP

Библиотека **OpenMP** [1,8] является стандартом для программирования на масштабируемых SMP-системах. В стандарт входят описания набора директив компилятора, переменных среды и процедур. За счет идеи «инкрементального распараллеливания» OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы директивы OpenMP.

Предполагается, что программа OpenMP на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть поставлены заглушки, текст которых приведен в спецификациях.

В OpenMP любой процесс состоит из нескольких нитей управления, которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки. В простейшем случае процесс состоит из одной нити.

Обычно для демонстрации параллельных вычислений используют простую программу вычисления числа π. Число π можно определить следующим образом:

$$\int_0^1 \frac{dx}{1+x^2} = arctg(1) - arctg(0) = \pi/4.$$

Вычисление интеграла затем заменяют вычислением суммы:

$$\int_0^1 \frac{4}{1+x^2}\,dx \approx \frac{1}{n}\sum_{i=1}^{n} \frac{4}{1+x_i{}^2},$$

где $x_i = 1/n(i - 0{,}5)$.

В последовательную программу вставляются две строки, и она становится параллельной.

```
program compute_pi
  parameter (n = 1000)
  integer i
  double precision w,x,sum,pi,f,a
  f(a) = 4.d0/(1.d0+a*a)
  w = 1.0d0/n
  sum = 0.0d0;
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+:sum)
  do i=1,n
     x = w*(i-0.5d0)
     sum = sum + f(x)
  enddo
  pi = w*sum
  print *,'pi = ',pi
  stop
  end
```

Программа начинается как единственный процесс на головном процессоре. Он исполняет все операторы до первой директивы типа PARALLEL. В данном примере это оператор PARALLEL DO, после исполнения которого следующий за ним цикл do ... enddo становится параллельным, т. е. в нем порождается множество процессов с соответствующим каждому процессу окружением.

В рассматриваемом примере окружение состоит из локальной переменной *x* (PRIVATE), переменной *sum* редукции (REDUCTION) и одной разделяемой переменной *w* (SHARED). Переменные *x* и *sum* локальны в каждом процессе без разделения между несколькими процессами. Переменная *w* располагается в головном процессе. Оператор редукции REDUCTION имеет в качестве атрибута операцию, которая применяется к локальным копиям параллельных процессов в конце каждого процесса для вычисления значения переменной в головном процессе. Переменная цикла *n* определяет общее количество итераций цикла. Эти итерации разбиваются на блоки смежных итераций и каждый блок назначается отдельному процессу. Параллельные процессы завершаются оператором enddo, выступающим как синхронизирую-

щий барьер для порожденных процессов. После завершения всех процессов продолжается только головной процесс.

Директивы OpenMP с точки зрения Фортрана являются комментариями и начинаются с комбинации символов «!$OMP». Директивы можно разделить на три категории: определение параллельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов – клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

Директивы определения параллельной секции PARALLEL приводят к порождению N–1 новых процессов, называемых в OpenMP нитями, а порождающая нить получает номер 0 и становится основной нитью команды («master thread»). При выходе из параллельной области основная нить дожидается завершения остальных нитей и продолжает выполнение в одном экземпляре. Значение N обычно равно числу процессоров в системе.

Блоки итераций между нитями (процессорами) могут распределяться по разному:

- **STATIC**, ***m*** – статически, блоками по m итераций;
- **DYNAMIC**, ***m*** – динамически, блоками по m (каждая нить берет на выполнение первый еще не взятый блок итераций);
- **GUIDED**, ***m*** – размер блока итераций уменьшается экспоненциально до величины m;
- **RUNTIME –** выбирается во время выполнения.

Директивы синхронизации обеспечивают как корректность работы нитей над общей памятью, так и барьерную синхронизацию, при которой каждая нить дожидается всех остальных, перед или после завершения параллельного участка программы.

Переменные OpenMP в параллельных областях программы разделяются на два основных класса: **SHARED** (общие – под именем A все нити видят одну переменную) и **PRIVATE** (приватные – под именем A каждая нить видит свою переменную).

2.1.2. Стандарт MPI

Система программирования **MPI** предназначена для ЭВМ с индивидуальной памятью, т. е. для многопроцессорных систем с обменом сообщениями. MPI имеет следующие особенности.

- MPI – библиотека функций, а не язык. Она определяет состав, имена, вызовы функций и результаты их работы. Программы, которые пишутся на языках FORTRAN, C и C++, компилируются обычными компиляторами и связаны с MPI-библиотекой.
- MPI – описание функций, а не реализация. Все поставщики параллельных компьютерных систем предлагают реализации MPI для своих машин бесплатно, реализации общего назначения также могут быть получены из Интернет. Правильная MPI-программа должна выполняться на всех реализациях без изменения.

В модели передачи сообщений процессы, выполняющиеся параллельно, имеют раздельные адресные пространства. Обмен происходит, когда часть адресного пространства одного процесса скопирована в адресное пространство другого процесса. Эта операция совместная и возможна только, когда первый процесс выполняет операцию передачи сообщения, а второй – операцию его получения.

Процессы в MPI принадлежат группам. Если группа содержит n процессов, то процессы нумеруются внутри группы номерами, которые являются целыми числами от 0 до n – 1. Имеется начальная группа, которой принадлежат все процессы в реализации MPI.

Контекст есть свойство коммуникаторов, которое позволяет разделять пространство обмена. Сообщение, посланное в одном контексте, не может быть получено в другом контексте. Контексты не являются явными объектами MPI, они проявляются только как часть реализации коммуникаторов.

Понятия контекста и группы объединены в едином объекте, называемом *коммуникатором*. Таким образом, отправитель или получатель, определенные в операции посылки или получения, всегда обращается к номеру процесса в группе, идентифицированной данным коммуникатором.

В MPI базисной операцией посылки является операция

MPI_Send(address, count, datatype, destination, tag, comm);

где **count** – количество объектов типа **datatype**, начинающихся с адреса **address** в буфере посылки; **destination** – номер получателя в группе, определяемой коммуникатором **comm**; **tag** – целое число, используемое для описания сообщения; **comm** – идентификатор группы процессов и коммуникационный контекст.

Базисной операцией приема является операция

MPI_Recv (**address, count, datatype, source, tag, comm, status**);

где **address**, **count**, **datatype** описывают буфер приемника, как в случае MPI_Send; **sourse** – номер процесса-отправителя сообщения в группе, определяемой коммуникатором **comm**; **status** – статус обмена. **Sourse**, **tag**, **count** фактически полученного сообщения можно определить на основе **status**. Например, если в качестве значения для переменной **source** используется значение MPI_ANY_SOURCE (любой источник), то определить номер фактического процесса отправителя можно через status.MPI_SOURCE.

В MPI используются коллективные операции, которые можно разделить на два вида:

- операции перемещения данных между процессами. Самая простая из них – широковещание. MPI имеет много и более сложных коллективных операций передачи и сбора сообщений;
- операции коллективного вычисления (минимум, максимум, сумма и другие, в том числе и определяемые пользователем операции).

В обоих случаях библиотеки функций коллективных операций строятся с использованием знания о преимуществах структуры машины, чтобы увеличить параллелизм выполнения этих операций.

Часто предпочтительно описывать процессы в проблемно-ориентированной топологии. В MPI используется описание процессов в топологии графовых структур и решеток.

В MPI введены средства, позволяющие определять состояние процесса вычислений, которые позволяют отладить программу и улучшить ее характеристики.

В MPI имеются как блокирующие операции обменов, так и неблокирующие их варианты, благодаря чему окончание этих операций может быть определено явно. MPI также имеет несколько коммуникационных режимов. Стандартный режим соответствует общей практике в системах передачи сообщений. Синхронный режим требует блокировать посылку на время приема сообщения в противоположность стандартному режиму, при котором посылка блокируется до момента захвата буфера. Режим по готовности (для посылки) – способ, предоставленный программисту, чтобы сообщить системе, что этот прием был зафиксирован. Следовательно, система более низкого уровня может использовать более быстрый протокол, если он доступен. Буферизованный режим позволяет пользователю управлять буферизацией.

Программы MPI могут выполняться на сетях машин, которые имеют различные длины и форматы для одного и того же типа дан-

ных, поэтому каждая коммуникационная операция определяет структуру и все компоненты типов данных. Следовательно, реализация всегда имеет достаточную информацию, чтобы делать преобразования формата данных, если они необходимы. MPI не определяет, как эти преобразования должны выполняться, разрешая реализации производить оптимизацию для конкретных условий.

Процесс есть программная единица, у которой имеется собственное адресное пространство и одна или несколько нитей. **Процессор** – фрагмент аппаратных средств, способный к выполнению программы.

Некоторые реализации MPI устанавливают, что в программе MPI всегда одному процессу соответствует один процессор; другие – позволяют размещать много процессов на каждом процессоре.

Если в кластере используются SMP-узлы, то для организации вычислений возможны два варианта.

1. Для каждого процессора в SMP-узле порождается отдельный MPI-процесс. MPI-процессы внутри этого узла обмениваются сообщениями через разделяемую память (необходимо настроить MPICH соответствующим образом).
2. На каждом узле запускается только один MPI-процесс. Внутри каждого MPI-процесса производится распараллеливание в модели «общей памяти», например с помощью директив OpenMP.

Чем больше функций содержит библиотека MPI, тем больше возможностей представляется пользователю для написания эффективных программ, но для написания подавляющего числа программ принципиально достаточно следующих шести функций:

MPI_Init – инициализация MPI;
MPI_Comm_size – определение числа процессов;
MPI_Comm_rank – определение процессом собственного номера;
MPI_Send – посылка сообщения;
MPI_Recv – получение сообщения;
MPI_Finalize – завершение программы MPI.

В качестве примера параллельной программы, написанной в стандарте MPI для языка C, рассмотрим программу вычисления числа π. Алгоритм вычисления π описан в предыдущем параграфе.

```
#include "mpi.h"
#include <math.h>
int main ( int argc, char *argv[ ] )
{   int n, myid, numprocs, i;
    double mypi, pi, h, sum, x, t1, t2, PI25DT = 3.14159265358979323846264 3;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    while (1)
    {  if (myid == 0)
       {      printf ("Enter the number of intervals: (0 quits) ");   scanf ("%d", &n);
              t1 = MPI_Wtime();
       }
       MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
       if (n == 0)  break;
       else
       {  h  = 1.0/ (double) n;
          sum = 0.0;
          for (i = myid +1; i <= n; i+= numprocs)
          {   x = h * ( (double)i - 0.5);
              sum += (4.0 / (1.0 + x*x));
           }
         mypi = h * sum;
         MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
         if (myid == 0)
         {  t2 = MPI_Wtime();
            printf ("pi is approximately %.16f. Error is %.16f\n",pi, fabs(pi - PI25DT));
            printf ("'time is  %f  seconds \n", t2-t1);
          }
       }
    }
  MPI_Finalize();
  return 0;
}
```

В программе после нескольких строк определения переменных следуют три строки, которые есть в каждой MPI-программе:

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Обращение к MPI_Init должно быть первым обращением в MPI-программе, оно устанавливает «среду» MPI. В каждой программе может выполняться только один вызов MPI_Init.

Коммуникатор MPI_COMM_WORLD описывает состав процессов и связи между ними. Вызов MPI_Comm_size возвращает в **numprocs** число процессов, которые пользователь задал при запуске программы. Процессы в любой группе нумеруются последовательными целыми числами, начиная с 0. Вызывая MPI_Comm_rank, каждый процесс выясняет свой номер (**myid**) в группе, связанной с коммуникатором

MPI_COMM_WORLD. Затем главный процесс (у которого **myid** = 0) получает от пользователя значение числа прямоугольников *n*:

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Первые три параметра соответственно обозначают адрес, количество и тип данных. Четвертый параметр указывает номер источника данных (головной процесс), пятый – название коммуникатора группы. Таким образом, после обращения к MPI_Bcast все процессы имеют значение *n* и собственные идентификаторы, что является достаточным для каждого процесса, чтобы вычислить **mypi** – свой вклад в вычисление π. Для этого каждый процесс вычисляет область каждого прямоугольника, начинающегося с **myid** + 1.

Затем все значения **mypi**, вычисленные индивидуальными процессами, суммируются с помощью вызова Reduce:

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
```

Первые два параметра описывают соответственно адреса источника и результата. Третий и четвертый параметры описывают число данных (*1*) и их тип, пятый – тип арифметико-логической операции, шестой – номер процесса для размещения результата.

Затем управление передается на начало цикла. Этим пользователю предоставляется возможность задать новое *n* и повысить точность вычислений. Когда пользователь печатает нуль в ответ на запрос о новом значении *n*, цикл завершается, и все процессы выполняют

```
MPI_Finalize();
```

после которого любые операции MPI выполняться не будут.

Функция MPI_Wtime() используется для измерения времени исполнения участка программы, расположенного между двумя включениями в программу этой функции.

2.1.3. Библиотека PETSc

К системным средствам программирования относятся библиотеки, содержащие наборы оптимизированных прикладных программ различного назначения. Существует большое количество библиотек, предназначенных для систем с распределенной памятью, в частности, использующих стандарт MPI. Среди них выделяются две библиотеки.

- ScaLAPACK (**Sca**lable **LAPACK**) – библиотека для параллельного решения разнообразных задач линейной алгебры [2].

• PETSc (**P**ortable **E**xtensible **T**oolkit for **S**cientific **C**omputation) – библиотека для параллельного решения линейных и нелинейных систем уравнений, возникающих при дискретизации уравнений в частных производных [1]. Это означает, что прикладные возможности библиотеки PETSc шире, чем возможности библиотеки ScaLAPACK. Кроме того, в PETSc максимально используется стандарт MPI. Одна из причин этого состоит в том, что в создании PETSc принимали участие разработчики стандарта MPI.

Описание фрагмента библиотеки PETSc представлено в III.

2.2. MPICH – ОСНОВНАЯ РЕАЛИЗАЦИИ MPI

Под реализацией обычно понимают программные средства, обеспечивающие выполнение всех функций MPI в исполнительной среде. *Исполнительная среда* – это то, что принадлежит локальной сети компьютеров: вычислительная и сетевая аппаратура, коммуникационные протоколы, операционная система. Программные средства реализации обеспечивают парные и коллективные обмены, реализуют коммуникаторы, топологию, средства общения с исполнительной средой, средства профилирования и множество вспомогательных операций, связанных, например, с запуском вычислений, оценкой эффективности параллельных вычислений и многое другое.

Реализации MPI разрабатываются как для операционных систем Unix (преимущественно для Linux), так и для Windows. Наибольшее распространение получили бесплатные реализации. Ниже перечислены некоторые из них:

• **MPICH** – переносимая реализация (работает почти на всех UNIX-системах и Windows NT), разработана в Аргоннской национальной лаборатории. Поддерживаются кластеры на базе SMP-узлов;

• **LAM** – реализация MPI для гетерогенных кластеров, т. е. кластеров, состоящих из процессоров с разной архитектурой и параметрами (университет Notre-Damme);

• **WMPI** – реализация MPI для платформ Win32 (Microsoft Windows 95/98/NT), разработанная в университете Coimbra (Португалия). Имеет малый объем и упрощенную инсталляцию;

• **MP-MPICH** – мультиплатформенная реализация MPI на базе MPICH. Включает NT-MPICH (версию MPICH для Windows NT) и SCI-MPICH (версию MPICH для SCI-коммутаторов). Разработка RWTH-Aachen (Аахен, Германия).

Некоторые коммерческие реализации предназначены для работы с ОС Windows:

- **MPI/PRO** for Windows NT – разработана компанией MPI Software Technology. Работает на кластерах рабочих станций и серверов Windows NT (платформы Intel и Alpha). Поддерживается стандарт MPI 1.2;
- **PaTENT MPI** – реализация MPI компании Genias Sotfware для кластеров на базе Windows NT;
- **PowerMPI –** реализация MPI для транспьютерных систем Parsytec.

Производители параллельных систем почти всегда поставляют оптимизированные для своих систем реализации интерфейса MPI, например для машин HP 9000, SGI/Cray, IBM SP2, Sun HPC и др. Информацию по системам можно найти на сайте *http://www.parallel.ru.*

2.2.1. Принципы построения MPICH

MPI – это описание библиотеки функций, которые обеспечивают в первую очередь обмен данными между процессами. Следовательно, чтобы такая библиотека работала в некоторой исполнительной среде, необходимо между описанием библиотеки и исполнительной средой иметь промежуточный слой, который называется *реализацией* MPI для данной исполнительной среды.

Возможны два способа построения реализаций: прямая реализация для конкретной ЭВМ и реализация через **ADI** (**A**bstract **D**evice **I**nterface – интерфейс для абстрактного устройства).

Поскольку имеется большое количество типов параллельных систем, то количество реализаций в первом случае будет слишком велико. Во втором случае строится реализация только для одного ADI, а затем архитектура ADI поставщиками параллельного оборудования реализуется в конкретной исполнительной среде (как правило, программно). Такая двухступенчатая реализация MPI уменьшает число вариантов реализаций и обеспечивает переносимость реализации. Поскольку аппаратно зависимая часть этого описания невелика, использование метода ADI позволяет создать пакет программ, который разработчики реальных систем могут использовать практически без переделок. В архитектуре ADI объединяются возможности перспективных параллельных ЭВМ.

Основной объем работ по разработке стандарта MPI и построению его реализаций выполняется в Аргоннской национальной лаборатории

[1]. Здесь подготовлены и получили широкое распространение реализации MPI, получившие название MPICH (добавка CH взята из названия пакета Chameleon, который ранее использовался для систем с передачей сообщений; многое из этого пакета вошло в MPICH). Имеются три поколения MPICH, связанные с развитием ADI.

Первое поколение ADI-1 было спроектировано для компьютеров MPP, где механизм обмена между процессами принадлежал системе. Этот ADI обеспечивал хорошие характеристики с малыми накладными расходами, такая версия была установлена на параллельных компьютерах: Intel iPSC/860, Delta, Paragon, nCUBE.

Второе поколение ADI-2 было введено, чтобы получить большую гибкость в реализациях и эффективно поддерживать коммуникационные механизмы с большим объемом функций.

Третье поколение ADI-3 было спроектировано, чтобы обеспечить поддержку для сетей с удаленным доступом, многопоточной исполнительной среды и операций MPI-2, таких как удаленный доступ к памяти и динамическое управление процессами. ADI-3 является первой версией MPICH, в которой при проектировании не ставилась задача близкого соответствия другим библиотекам с обменом сообщениями (в частности, PVM), поскольку MPI вытеснил большинство систем с обменом сообщениями в научных вычислениях. ADI-3 подобно предыдущим ADI спроектирован так, чтобы содействовать переносу MPICH на новые платформы и коммуникационные среды.

ADI для обмена сообщениями должен обеспечивать следующие функции:

- описание передаваемых и получаемых сообщений;
- перемещение данных между ADI и передающей аппаратурой;
- управление списком зависших сообщений (как посланных, так и принимаемых);
- получение основной информации об исполнительной среде и ее состоянии (например, число выполняемых задач).

Следовательно, ADI – это совокупность определений функций (которые могут быть реализованы как функции или макроопределения в языке C) из пользовательского набора MPI. Если так, то это создает протоколы, которые отличают MPICH от других реализаций MPI. В частности, уровень ADI содержит процедуры для упаковки сообщений и подключения заголовочной информации, управления политикой буферизации, для установления соответствия приемов приходящих сообщений и др.

Для того чтобы понять, как работает MPICH, в качестве примера рассмотрим реализацию простых функций посылки и приема MPI_Send и MPI_Recv. Для этой цели можно использовать два протокола: Eager и Rendezvous.

Eager. При посылке данных MPI вместе с адресом буфера должен включить информацию пользователя о тэге, коммуникаторе, длине, источнике и получателе сообщения. Эту дополнительную информацию называют *оболочкой* (envelope). Посылаемое сообщение состоит из оболочки, которая следует за данными. Метод посылки данных вместе с оболочкой называется «*жадным*» (*eager*) *протоколом*.

Когда сообщение прибывает, возможны два случая: соответствующая приемная процедура запущена либо нет. В первом случае предоставляется место для приходящих данных. Во втором случае ситуация сложнее. Принимающий процесс должен помнить, что сообщение прибыло, и где-то его сохранить. Первое требование выполнить относительно легко, отслеживая очередь поступивших сообщений. Когда программа выполняет MPI_Recv, она прежде всего проверяет эту очередь. Если сообщение прибыло, операция выполняется и завершается. Но с данными может быть проблема. Что, например, будет, если множество подчиненных процессов почти одновременно пошлют главному процессу свои длинные сообщения (например, по 100 Мбайт каждое) и места в памяти главного процесса для их размещения не хватит? Стандарт MPI требует, чтобы прием данных выполнялся, а не выдавался отказ. Это и приводит к буферизации.

Rendezvous. Чтобы решить проблему доставки большого объема данных по назначению, нужно контролировать, как много и когда эти данные прибывают в процесс-получатель.

Одно простое решение состоит в том, чтобы послать процессу-получателю только оболочку. Когда процесс-получатель потребует данные (и имеет место для их размещения), он посылает отправителю сообщение, в котором говорится: «теперь посылай данные». Отправитель затем пошлет данные, с уверенностью, что они будут приняты. Такой метод называется *протоколом* «*рандеву*» (*Rendezvous*). В этом случае получатель должен отводить память только для хранения оболочек, имеющих очень небольшой размер, а не для хранения самих данных.

Причина для разработки многих вариаций режимов передачи теперь довольно ясна. Каждый режим может быть реализован с помо-

щью комбинации «жадного» и «рандеву» протоколов. Некоторые варианты представлены в табл. 2.1.

Таблица 2.1.

Режимы send с протоколами Eager и Rendezvous

Вызов MPI	Размер сообщения	Протокол
MPI_Ssend	any	Rendezvous всегда
MPI_Rsend	any	Eager всегда
MPI_Send	≤16 Кбайт	Eager
MPI_Send	>16 КБайт	Rendezvous

Главное преимущество протокола «рандеву» состоит в том, что он позволяет принимать произвольно большие сообщения в любом количестве. Но этот метод невыгодно использовать для всех сообщений, поскольку, например, «жадный» протокол быстрее, в частности для коротких сообщений. Возможно большое количество других протоколов и их модификаций.

Канальный интерфейс является одним из наиболее важных уровней иерархии ADI и может иметь множественные реализации.

На рис. 2.1 имеются два ADI: p4 – для систем с передачей сообщений и p2 – для систем с разделяемой памятью. Реализация Chameleon создана давно, построена на базе интерфейса p4, некоторые ее элементы использовались на начальной стадии реализации MPICH. В MPICH также используется интерфейс p4, который перенесен поставщиками аппаратуры на ряд машин, что и показано на рис. 2.1. Интерфейс p2 адаптирован для ряда систем.

Однако на рис. 2.1 представлены и примеры прямой реализации MPI без промежуточного ADI. Так, ряд машин напрямую использует макросы, из которых состоит сама реализация Chameleon.

Другим примером непосредственной реализации канального интерфейса являются машины, использующие коммуникационные сети SCI. Рис. 2.1. характеризует гибкость в построении канального интерфейса. Это, в частности, относится к машинам SGI. MPICH стал использоваться на машинах SGI с самого начала. Это отмечено блоком SGI(0). Затем стала использоваться усовершенствованная версия SGI(1), использующая интерфейс разделяемой памяти. SGI (2) является прямой реализацией канального интерфейса, использующей ряд новых механизмов. Затем появились еще более совершенные варианты SGI(3) и SGI(4), которые обходят канальный интерфейс и ADI.

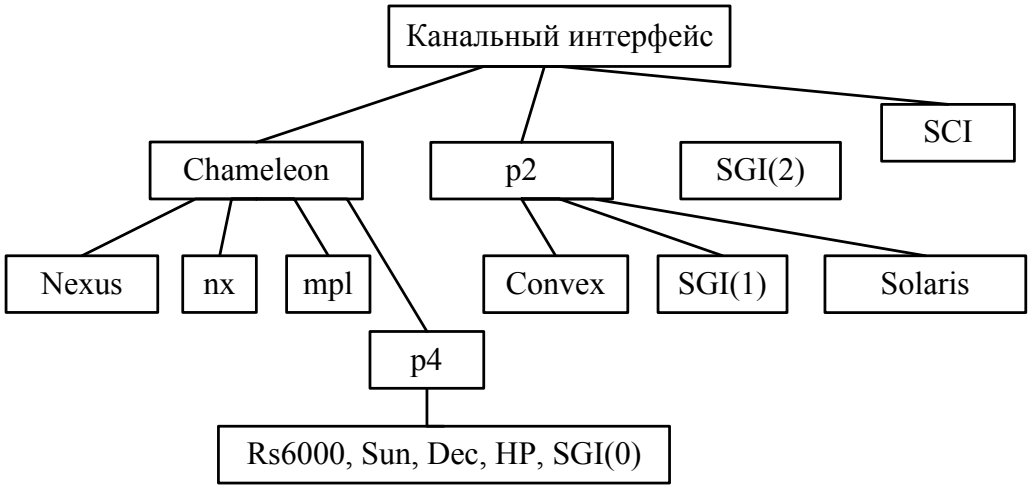


Рис. 2.1. Нижние слои MPICH

MPI в зависимости от версии содержит 150–200 функций, и все они должны быть реализованы с помощью MPICH. Часто реализация полного набора функций растянута во времени и занимает длительный период.

MPICH является переносимой реализацией полного описания MPI для большого числа параллельных вычислительных сред, включая кластеры рабочих станций и массово-параллельные системы. MPICH, кроме самой библиотеки функций MPI, содержит программную среду для работы с программами MPI. Программная среда включает мобильный механизм запуска, несколько профилирующих библиотек для изучения характеристик MPI-программ и интерфейсов для всех других средств.

2.2.2. Управление запуском процессов

Для запуска процессов используется программа многоцелевого демона **mpd** (**m**ulti**p**urpose **d**aemon). Целью создания mpd и связанного с ним устройства ADI ch_p4mpd является необходимость обеспечить поведение программы, реализующей команду запуска (mpirun) как единого целого, даже если она создает множество процессов для выполнения задачи MPI.

Далее будем различать процесс mpirun и процессы MPI. Поведение программы mpirun включает в себя следующие функции:

- быстрый масштабируемый запуск процессов MPI;
- сбор вывода и сообщений об ошибках от процессов MPI в файлы стандартного вывода и сообщений об ошибках процесса mpirun;

- доставку стандартного ввода mpirun на стандартный ввод процесса MPI с номером 0;
- доставку сигналов от процесса mpirun ко всем процессам MPI. Это означает прекращение, приостановку и возобновление программы параллельной задачи, как будто она является единым целым;
- доставку аргументов командной строки ко всем процессам MPI.

В прежних системах запуска устройство ADI ch_p4 по умолчанию обращалось к команде удаленного доступа (rsh) в Unix для обработки запуска на удаленной машине. Необходимость идентификации во время запуска задачи, сочетающаяся с последовательным процессом, в котором информация собирается с каждой удаленной машины и отсылается широковещательно всем остальным, делало запуск задачи неприемлемо медленным, особенно для большого числа процессов.

Основной идеей ускорения запуска является создание перед моментом запуска задачи сети демонов на машинах, которые будут исполнять процессы MPI, а также на машине, где будет выполняться команда mpirun. После этого команды запуска задачи (и другие команды) будут обращаться к локальному демону и использовать уже существующие демоны для запуска процессов. Большая часть начальной синхронизации, выполняемой устройством ADI ch_p4, устраняется, поскольку демоны используются для поддержания установленных соединений между процессами. Как только демоны запускаются, они соединяются в кольцо: «консольный» процесс (mpirun, mpdtrace и др.) может соединиться с любым mpd через именованный сокет Unix, установленный в */tmp* локальным mpd. *Сокет* – это совокупность адреса узла сети и номера одного из программных портов этого узла. Если это процесс mpirun, он требует, чтобы было запущено определенное число процессов, начиная с заданной машины. По умолчанию место размещения – следующий mpd в кольце после того, который был запрошен с консоли.

После этого происходят следующие события (рис. 2.2):

- mpd порождает требуемое количество процессов-менеджеров; исполняемый файл называется mpdman и располагается в каталоге *mpich/mpid/mpd*;
- менеджеры самостоятельно объединяются в кольцо и порождают процессы приложения, называемые клиентами;
- консоль отсоединяется от mpd и присоединяется к первому менеджеру, stdin от mpirun доставляется к клиенту менеджера 0;

• менеджеры перехватывают стандартный ввод-вывод от клиентов и доставляют им аргументы командной строки и переменные окружения, заданные в команде mpirun. Сокеты, содержащие stdout и stderr формируют дерево с менеджером 0 в качестве корня.

С этого момента ситуация выглядит подобно представленной на рис. 2.2. Когда клиенту необходимо соединиться с другим клиентом, он использует менеджеры, чтобы найти подходящий процесс на машине-приемнике. Процесс mpirun может быть приостановлен. В этом случае останавливаются и его клиенты, однако mpd и менеджеры продолжают выполняться, чтобы разбудить клиентов после пробуждения mpirun. Уничтожение mpirun уничтожает и клиентов, и менеджеров.

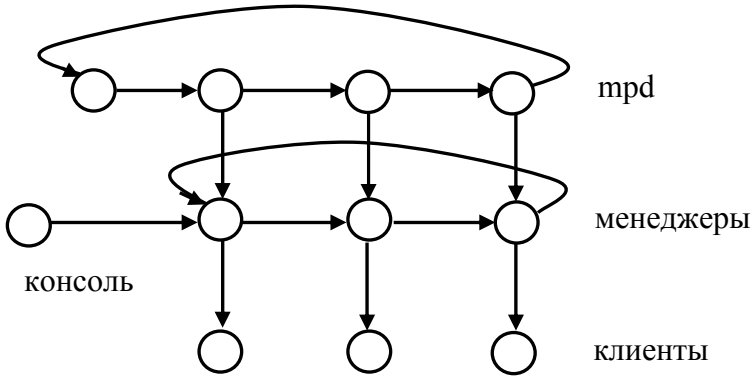


Рис.2.2. Система mpd с консолью, менеджерами и клиентами

Одно и то же кольцо mpd может использоваться для запуска множества задач с множества консолей в одно и то же время. При обычных условиях необходимо, чтобы для каждого пользователя существовало отдельное кольцо. Для безопасности каждый пользователь должен иметь в своем домашнем каталоге доступный для чтения только ему файл mpdpasswd, содержащий пароль. Файл будет считываться при запуске mpd. Только mpd, знающие этот пароль, могут войти в кольцо существующих mpd.

Поскольку демоны mpd уже находятся в соединении друг с другом перед запуском задачи, то запуск происходит гораздо быстрее, чем при использовании устройства ADI ch_p4. Команда mpirun для устройства ch_p4mpd имеет ряд специальных вариантов.

2.2.3. Состав пакета MPICH и назначение его элементов

Пакет MPICH – реализация системы параллельного программирования MPI. Старшие версии MPICH полностью совместимы со спецификацией стандарта MPI-1, частично поддерживают MPI-2, частично параллельный ввод-вывод (ROMIO), содержат средства профилирования (сбора характеристик процесса вычислений) и средства визуализации производительности параллельных программ, примеры тестов производительности и проверки функционирования системы.

К недостаткам пакета (и стандарта MPI-1 в целом) можно отнести невозможность запуска процессов во время исполнения программы и отсутствие средств мониторинга текущего состояния вычислительной системы. В итоге сбой на одном из узлов вычислительной системы приводит к аварийному завершению MPI-программы, а невозможность динамического подключения и отключения узлов не позволяет максимально эффективно использовать ресурсы системы.

MPICH для ОС Windows NT поставляется в двух версиях: с автоматической инсталляцией и с ручной установкой на основе исходных кодов программ. Структура каталогов в этих поставках различается.

Рассмотрим версию с ручной установкой, как более полную. В качестве примера будем использовать пакет mpich.nt.1.2.5. Пакет поставляется в файлах *mpich.nt.1.2.5.src.exe* или *mpich.nt.1.2.5.src.zip* [1].

Следует отметить, что различные версии MPICH исторически создавались прежде всего для реализаций ОС Unix, например Linux. Реализации для ОС Windows NT появились значительно позже. При этом все новое, что разрабатывалось специально для Windows NT, просто добавлялось в существующие папки, поэтому в состав пакета MPICH входят средства, необходимые и для ОС Linux, и для ОС Windows. Далее мы опишем средства, необходимые преимущественно для работы с Windows NT.

Распакованный архив MPICH состоит из следующих папок, которые содержат:

- **doc** – документацию по установке и использованию MPICH;
- **examples/nt** – поддиректории с примерами простых программ и тесты для проверки правильности установки (TEST) и для проверки производительности кластера (PERFTEST);
- **include** – h-файлы для MPI-программ, подключаемые компилятором;
- **lib** – lib-файлы для компоновки программ;

- **bin** – файлы, используемые для запуска программ, например файл mpirun;
- **mpe** – исходные коды библиотеки MPE, используемые для профилирования и выполнения графических операций; программы визуализации логфайлов производительности (например jumpshot3);
- **mpid** – исходные коды всех функций MPI и вспомогательные процедуры, написанные для различных ADI (p4, ch_p4, ch_p4dem и др.), выбираемых для конкретной исполнительной среды;
- **romio** – исходные коды реализации параллельной системы ввода-вывода ROMIO; не следует путать ROMIO с реализацией RMO (Remout Memory Operations), которая используется в MPI-2 для того, чтобы скрыть от пользователя низкоуровневые операции обмена типа Send и Recv; обмен в случае RMO осуществляется с помощью обычных обращений к памяти (load from, store into), а система сама решает, является ли некоторое обращение локальным или удаленным;
- **src** – исходные коды всех функций MPI и вспомогательные процедуры MPICH; в этих кодах находятся обращения к функциям соответствующих приборов из папки MPID, которые и обеспечивают выход на коммуникационные протоколы NetBIOS или TCP/IP;
- **www** – HTML-версию описания функций. Файл index.html содержит ссылки на список функций MPI, а также на документацию по установке и настройке MPICH.

В директории **bin** содержатся следующие программы:

- **clog2slog** – конвертирует логфайлы формата CLOG в формат SLOG, генерируемые при профилировании параллельных программ; с помощью этих логфайлов анализируется эффективность параллельных вычислений;
- **guimpijob** –отображает текущие MPI-задания, выполняемые на кластере в графическом интерфейсе;
- **guimpirun** – графический интерфейс для запуска программ MPI;
- **mpd** – сервис NT, позволяющий производить запуск MPI-программ; в данной реализации к нему обращается mpirun при запуске программ на кластере;
- **mpdupdate** – утилита, позволяющая обновлять mpd;
- **mpiconfig** – программа для конфигурации вычислительной сети с графическим интерфейсом, позволяет получить имена машин для использования в качестве узлов вычислительной сети, задать таймаут, сделать другие настройки;

- **mpijob** – позволяет просмотреть текущие MPI-задания, выполняемые на кластере;
- **mpiregister** – утилита, позволяющая установить пользователя, с правами которого будут запускаться MPI-программы; в ОС Windows настройки сохраняются в реестре для конкретного пользователя, для разных пользователей настройки могут быть разными;
- **mpirun** – утилита для запуска MPI-программ из командной строки. В качестве параметров принимает: число процессов, следующее за ключом -np или -localonly; имя запускаемой программы; другие параметры либо имя файла конфигурации, написанного в специальном формате. Сведения о форматах командной строки и файла конфигурации можно получить, запустив mpirun без параметров либо с параметром –help.

Директория **mpe** содержит следующие элементы:

- **clog2slog** – исходный код утилиты преобразования из формата **clog** в формат **slog**;
- **include** – подключаемые компилятором файлы MPE;
- **profiling** – систему визуализации (для UNIX);
- **slog_api** – библиотеку функций для генерации логфайлов формата SLOG; содержит свои папки: **doc** с документацией, **include** с подключаемыми файлами, **src** с исходными кодами;
- **src** – исходный код MPE.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое стандарт OpenMP?
2. Опишите, как выполняется в языке OpenMP программа вычисления числа π.
3. Назовите типы директив стандарта OpenMP.
4. Какие классы переменных используются в OpenMP?
5. Что такое стандарт MPI?
6. Назовите основные операции передачи и приема в MPI.
7. Назовите и опишите состав и назначение параметров обменных функций MPI.
8. Что такое процесс и процессор в MPI?
9. Перечислите минимально возможный состав MPI функций.
10. Расскажите, как выполняется программа MPI для вычисления числа π.
11. Какой коммуникатор определен после выполнения функции MPI_Init?
12. Можно ли использовать функции MPI до вызова MPI_Init?
13. Как в MPI определить номер процесса?
14. Как узнать число запущенных процессов приложения?
15. Возможна ли замена в MPI коллективных операций на парные обмены?
16. Что такое библиотека PETSc?
17. Что такое реализация MPI?

18. Какие реализации MPI Вы знаете?
19. Каковы функции MPICH?
20. Для чего MPICH строится через ADI?
21. Что такое ADI и какие ADI вы знаете?
22. Какие протоколы используются для реализации операций посылки и приема?
23. Чем протокол Eager отличается от протокола Rendezvous?
24. Как организован канальный интерфейс для различных приборов?
25. Как организован запуск процессов в MPICH?
26. Что такое MPD?
27. Опишите состав пакета MPICH.
28. Для чего используется папка bin?
29. Что размещено в папке MPID?

ГЛАВА 3. ИНСТАЛЛЯЦИЯ И НАСТРОЙКА СРЕДСТВ КЛАСТЕРА

3.1. ТРЕБОВАНИЯ К АППАРАТНОМУ ОБЕСПЕЧЕНИЮ КЛАСТЕРА

При комплектации кластера необходимо выполнить ряд общих требований.

1. Наиболее важным требованием является надежность оборудования (в частности компьютеров), так как MPI-программа запускается на нескольких компьютерах сети и выход из строя хотя бы одного из них приводит к некорректному завершению программы. Если вероятность безотказной работы одного из узлов распределенной вычислительной системы равна p_i, то вероятность безотказной работы всей системы q будет выражаться как произведение вероятностей: $q = \prod_i p_i$, т. е. q уменьшается пропорционально количеству узлов.

   Например, если средняя наработка одного компьютера на отказ составляет пять суток, то наработка кластера из 16 таких компьютеров составит 8 ч. Этого недостаточно, поскольку задачи на кластерах решаются непрерывно на протяжении многих часов и суток, поэтому при выборе компьютеров для кластера надо обращать внимание не только на максимальную тактовую частоту (это важно для автономного использования компьютеров), но и на надежность компьютеров, состоятельность компании-изготовителя.
2. При прочих равных условиях следует выбирать компьютеры с наибольшей тактовой частотой, иначе очередная модель микропроцессора Intel окажется более производительной, чем кластер небольшого размера.

3. При прочих равных условиях также следует выбирать компьютеры с наибольшим объемом кэша, объемом и быстродействием оперативной памяти, поскольку эти характеристики существенно влияют на время выполнения параллельной программы.
4. Наконец, для выполнения обменов в сети следует применять не концентратор (hub), в котором все обмены выполняются последовательно через общую шину, а коммутатор, поскольку он обеспечивает параллельное выполнение операций обмена и, следовательно, значительно увеличивает масштабируемость решения задач.

3.2. ИНСТАЛЛЯЦИЯ И НАСТРОЙКА ЭЛЕМЕНТОВ КЛАСТЕРА

Комплекс системных программных средств, включающий среду проектирования Visual C++, пакет MPICH, представляет собой необходимый набор элементов для создания и запуска параллельных MPI-программ. Для написания параллельных программ с использованием параллельной библиотеки можно установить библиотеку PETSc для Windows-кластера. Библиотека PETSc реализует распространенные операции матричной алгебры с использованием MPI и может быть сконфигурирована с реализацией MPICH для Windows.

Среда разработки VC++ поставляется с программой установки. В ОС Windows, в отличие от UNIX-систем, компилятор является частью IDE (среды разработки), а не самой ОС. Поэтому после установки необходимо совершить некоторые дополнительные действия, чтобы компилятором от Microsoft могли пользоваться и другие программы, не только VC++. Для этого нужно прописать пути в файловой системе к компилятору и библиотекам в переменные path, include и lib, как будет показано далее.

3.2.1. Установка MPICH

Описание производится на примере пакета MPICH 1.2.5 (адрес сайта *http://www-unix.mcs.anl.gov/mpi/mpich*/). Этот пакет для ОС Windows поставляется в двух версиях:

- в полной версии с исходными кодами;
- в версии с автоматической установкой без библиотеки профилирования (MPE).

Остановим выбор на полной версии. Разархивируем содержимое в локальную папку, например *C:\Apps\mpich*. После этого пропишем пути к программам и подключаемым файлам для всех пользователей кластера, либо для системы в целом.

Необходимо добавить пути (path) к подключаемым файлам и программам библиотеки MPICH в переменные path, include и lib. Для нашего примера пути описаны в табл. 3.1.

Таблица 3.1.

Пути к подключаемым файлам

include	C:\Apps\Microsoft Visual Studio\VC98\atl\include; C:\Apps\Microsoft Visual Studio\VC98\mfc\include; C:\Apps\Microsoft Visual Studio\VC98\include; C:\Apps\mpich\include;C:\Apps\mpich\mpe\include
lib	C:\Apps\Microsoft Visual Studio\VC98\mfc\lib; C:\Apps\Microsoft Visual Studio\VC98\lib; C:\Apps\mpich\lib
path	%SystemRoot%\system32; %SystemRoot%; %SystemRoot%\System32\Wbem; C:\Apps\mpich\bin; C:\Apps\mpich\lib; C:\Apps\Microsoft Visual Studio\VC98\Bin; C:\Apps\Microsoft Visual Studio\Common\Tools; C:\Apps\Microsoft Visual Studio\Common\MSDev98\Bin; C:\Apps\Microsoft Visual Studio\Common\Tools\WinNT

Обратим внимание, что у MPICH в переменную path попадают две папки: bin и lib, так как в lib содержится библиотека mpich.dll, необходимая для работы MPICH-программ.

При использовании ОС Windows для этого можно воспользоваться интерфейсом (рис. 3.1, 3.2). Окно «Свойства системы» может быть вызвано из «Панели управления» Windows (рис. 3.1). Вызвав режим «Переменные среды» (Environment Variables), можно увидеть окно, которое позволяет установить системные переменные (рис. 3.2).

Далее необходимо установить **mpd** в качестве системного сервиса, вызвав из папки *C:\Apps\mpich\bin* команду

mpd -install -interact

Чтобы получить список машин, которые будут использоваться для запуска MPI-программ, нужно сконфигурировать mpd, вызвав его с опцией –console:

mpd -console

либо с помощью программы mpiconfig без параметров:

mpiconfig

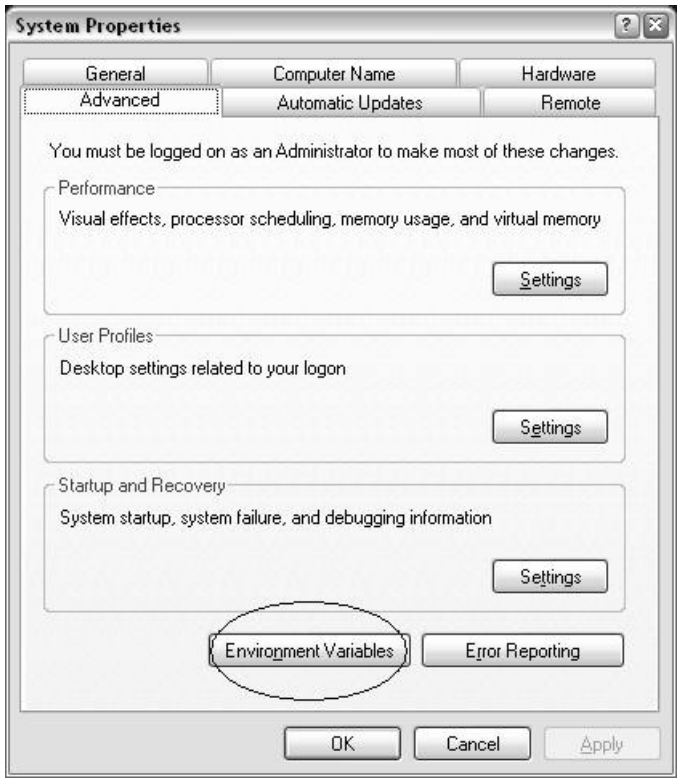


Рис. 3.1. Окно «Свойства системы»
для установки переменных среды

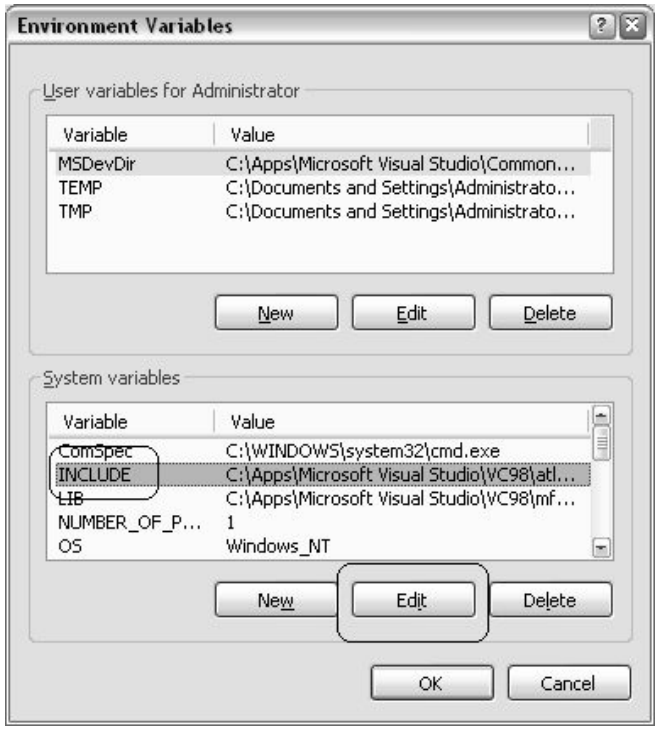


Рис. 3.2. Окно «Переменные среды»
для описания необходимых системных переменных

При вызове этой программы появляется окно, в котором необходимо выбрать кнопку «Select», затем в появившемся диалоге режим «Action/Scan hosts». В результате появится список доступных машин кластера. Нужно установить этот список в переменную hosts в mpd («Apply all»). Для установки пользователя, с правами которого запускаются MPI-программы, существует команда

mpiregister

без параметров. Она запросит имя и пароль пользователя, а затем запросит подтверждение на внесение данных. Данные с настройками mpd хранятся в реестре Windows в ключе HKEY_LOCAL_MACHINE\SOFTWARE\MPICH\MPD, т. е. едины для всех пользователей данной машины, в отличие от данных о пользователе, которые сохраняются в ключе реестра HKEY_CURRENT_USER\Software\MPICH и, следовательно, для каждого пользователя свои. Теперь, перезагрузив ОС Windows, можно проверить правильность установки MPICH, запустив примеры, как описано в гл. 4.

3.2.2. Установка PETSc

Библиотека PETSc доступна по адресу *http://www.mcs.anl.gov/petsc*. Поставляется для ОС UNIX, но существует возможность ее установить для ОС Windows, применяя некоторые дополнительные средства.

Установка выполняется следующим образом. Разархивируем библиотеку в локальную папку, например в *C:\Apps\petsc*. Для инсталляции библиотеки необходимо иметь версию пакета CygWin. Этот пакет включает реализацию некоторых UNIX-программ для Windows. Для PETSc-библиотеки необходимы как минимум следующие файлы: ash, cygwin, diff, fileutils, grep, make, sed, sh-utils, textutils. Пакет доступен по адресу *http://sources.redhat.com/cygwin*. Установим его в директорию *C:\Apps\cygwin*, пропишем пути для папок path и include. Далее установим библиотеку с оптимизированными для данного процессора функциями линейной алгебры Blas/Lapack, например библиотеку MKL и пропишем необходимые пути. MKL можно найти по адресу *http://developer.intel.com/software/products/mkl/mkldownload_new.htm*.

Непосредственно перед компиляцией библиотеки PETSc должны быть установлены также системные переменные PETSC_ARCH и PETSC_DIR. Для нашего случая это – *win32_ms_mpich* и *c:/apps/petsc* (важно использовать именно такую форму записи) соответственно. Все необходимые описания приведены в табл. 3.2. После установки всех переменных необходимо перезагрузить систему.

Таблица 3.2

Установка необходимых путей

include	C:\Apps\petsc\include; C:\Apps\petsc\bmake\win32_ms_mpich; C:\Apps\cygwin\usr\include; C:\Apps\Intel\MKL\include
lib	C:\Apps\Intel\MKL\ia32\lib
path	C:\Apps\cygwin\bin; C:\Apps\petsc\bin; C:\Apps\Intel\MKL\ia32\bin
PETSC_ARCH	win32_ms_mpich
PETSC_DIR	c:/apps/petsc

Теперь компилируем библиотеку PETSc на каждой машине кластера. Для этого выполняем следующие команды:

```
cd c:\apps\petsc
make BOPT=g all
make BOPT=g testexamples_uni
```

При этом не должно возникать никаких ошибок, иначе библиотека не будет установлена. Компиляция может занимать более 5 часов, это одна из наиболее ресурсоемких задач, которые довольно часто приходится решать.

После того как библиотека PETSc установлена, добавим путь *C:\Apps\petsc\lib\libg\win32_ms_mpich* в переменную lib.

Проверить правильность установки можно с помощью стандартной PETSc-программы, например из директории *%petsc%\projects\c\sles*, где *%petsc%* – директория, куда был установлен PETSc. Как компилировать и запускать программы, описано в параграфе 4.6. В параметры компоновщику следует добавить

```
mkl_p3.lib mkl_lapack.lib mpich.lib /nodefaultlib:"LIBCMTD.lib"
```

3.3. ТЕСТИРОВАНИЕ КЛАСТЕРА НА РАБОТОСПОСОБНОСТЬ И ПРОИЗВОДИТЕЛЬНОСТЬ

Под работоспособностью упрощенно понимается возможность выполнения на кластере функций MPI, а под производительностью – время их выполнения. Для определения работоспособности кластера необходимо выполнить следующие действия.

- Проверить доступность всех машин в локальной сети. Для этого используются обычные для локальной сети средства. Доступность отдельной машины можно проверить командой

ping <имя машины>

- Проверить доступность всех машин в кластере (сеть MPICH), созданном на базе локальной сети. Это можно выполнить, например, с помощью команды mpiconfig**,** расположенной в папке *bin* пакета MPICH. Если некоторые машины в сети MPICH недоступны, следует проверить правильность инсталляции и настройки пакета MPICH на недоступной машине. Недоступность может быть связана с тем, что нет доступа к mpd, либо программа не может найти требуемые библиотеки на данном узле.
- Запустить любую программу, написанную с использованием функций MPI. Конечно, для полной проверки работоспособности кластера надо запускать специальные тесты, которые имеются, например, в папке *example* пакета MPICH. Для этой цели можно использовать также описываемые далее тесты производительности.

Следует отметить, что проверка работоспособности кластера является относительно простой задачей. Гораздо сложнее обстоит дело с анализом производительности кластера. На производительность кластера влияют:

1. свойства задачи – ее потенциальный параллелизм, объемы счета и коммуникаций, объемы передаваемых пакетов данных и др.;
2. характеристики аппаратуры, системного обеспечения, их адекватность решаемой задаче и настройка;
3. правильность использования кластера, особенно при многопользовательском доступе.

Свойства задач – переменный фактор для кластера, пункты 2 и 3 определяют устойчивые параметры кластера. Для их определения предназначены описываемые ниже тесты производительности. Рассматриваемые тесты производительности разработаны в НИВЦ МГУ и доступны по адресу [2] в виде исходных текстов на языке С и полного описания. Разработанный пакет включает четыре теста, три из них тестируют эффективность сети передачи данных между узлами кластера, а четвертый – производительность совместного доступа узлов к сетевому файл-серверу:

- **transfer** – тест латентности и скорости пересылок между узлами кластера;

- **nettest** – тест пропускной способности сети при сложных обменах по различным логическим топологиям;
- **mpitest** – тест эффективности основных операций MPI;
- **nfstest** – тест производительности файл-сервера.

3.3.1. Тест transfer

Тест производительности межпроцессорных обменов предназначен для измерения пиковых характеристик обмена данными между двумя узлами – латентности и пропускной способности.

Латентностью (задержкой) называется время, затрачиваемое программным обеспечением и устройствами сети на подготовку к передаче информации по данному каналу. Полная латентность складывается из программной и аппаратной составляющих, измеряется в мкс.

Под *пропускной способностью* сети будем понимать количество информации, передаваемой между узлами сети в единицу времени, измеряется в МБайт/с. Очевидно, что реальная пропускная способность снижается программным обеспечением за счет передачи разного рода служебной информации.

Различают следующие виды пропускной способности сети:

- пропускная способность однонаправленных пересылок («точка-точка», uni-directional bandwidth), равная максимальной скорости, с которой процесс на одном узле может передавать данные другому процессу на другом узле;
- пропускная способность двунаправленных пересылок (bi-directional bandwidth), равная максимальной скорости, с которой процессы могут одновременно обмениваться данными по сети.

Время $T(L)$, необходимое на передачу сообщения длины L, можно определить следующим образом:

$$T(L) = s + L / R,$$

где T – полное время передачи сообщения; s – латентность; L – длина сообщения; а R – пропускная способность канала связи.

Для измерения пропускной способности режима «точка-точка» используется следующая методика. Процесс с номером 0 посылает процессу с номером 1 сообщение длины L байтов. Процесс 1, приняв сообщение от процесса 0, посылает ему ответное сообщение той же длины. Используются блокирующие вызовы (MPI_Send, MPI_Recv). Эти действия повторяются N раз с целью минимизировать погрешность за счет усреднения. Процесс 0 измеряет время T, затраченное на

все эти обмены. Пропускная способность R определяется по формуле $R = 2NL / T$.

Пропускная способность двунаправленных обменов определяется по той же формуле. При этом используются неблокирующие вызовы (MPI_Isend, MPI_Irecv). Производится измерение времени, затраченного процессом 0 на передачу сообщения процессу 1 и прием ответа от него при условии, что процессы начинают передачу информации одновременно после точки синхронизации.

Латентность измеряется как время, необходимое на передачу сигнала или сообщения нулевой длины. При этом для снижения влияния погрешности и низкого разрешения системного таймера, важно повторить операцию посылки сигнала и получения ответа большое число N раз. Таким образом, если время на N итераций пересылки сообщений нулевой длины туда и обратно составило T с, то латентность измеряется как $s = T / (2N)$.

Тест можно запускать и на большем количестве процессоров, в таком случае будет производиться последовательное тестирование производительности обменов между головным и всеми остальными процессорами. Тест реализован в виде нескольких файлов на языке C.

3.3.2. Тест nettest

Этот тест предназначен для измерения коммуникационной производительности при сложных обменах между несколькими узлами в различных логических топологиях. Тест может использоваться для проверки «выживаемости» сетевого оборудования при пиковых нагрузках и для определения пиковой пропускной способности коммутатора. Описанные методики тестирования обобщаются для нескольких процессов.

При этом рассматриваются три логические топологии взаимодействия процессов:

- звезда – взаимодействие главного процесса с подчиненными;
- кольцо – взаимодействие каждого процесса со следующим;
- полный граф – взаимодействие каждого процесса с каждым.

Под логическим каналом будем подразумевать неупорядоченную пару узлов (A, B), которые в данной топологии могут обмениваться сообщениями. Логическая топология полностью определяется множеством задействованных логических каналов.

Для каждой топологии рассматриваются однонаправленные и двунаправленные обмены. По каждому логическому каналу между узла-

ми *A* и *B* информация в ходе теста передается в обе стороны: от узла *A* к узлу *B* и обратно передается по *L* байтов информации. Однако в случае однонаправленных обменов один из узлов, например *B*, ждет получения сообщения от *A*, и только тогда может передавать *A* свое сообщение. Во втором случае информация может передаваться в обе стороны одновременно.

Конкретные способы организации пересылок средствами MPI (блокирующие, неблокирующие, и другие пересылки) здесь не регламентируются; предполагается, что из всех соответствующих по семантике данной топологии и способу обменов будет выбран вариант с наименьшими накладными расходами.

3.3.3. Тест mpitest

Тест предназначен для определения эффективность реализации и выполнения некоторых базовых конструкций MPI в рамках тестируемого программно-аппаратного комплекса. Тестируемые конструкции представлены в табл. 3.3.

Замеры времени осуществляются по следующей схеме:

- производится синхронизация с помощью MPI_Barrier;
- замеряется текущее время t_1;
- исполняются вызовы, относящиеся к тестируемой конструкции;
- снова производится барьерная синхронизация;
- замеряется текущее время t_2;
- время, затраченное на исполнение тестируемой конструкции, вычисляется как $t = t_2 – t_1 – DT – DB$.

В целях минимизации погрешности эта процедура повторяется несколько раз. Величина t усредняется по всем итерациям.

3.3.4. Тест nfstest

Это тест производительности общей файловой системы с точки зрения параллельных MPI-программ. Тест также можно использовать для проверки корректности функционирования файл-сервера при больших нагрузках. Данный тест не является тестом производительности MPI, но использует некоторые возможности MPI, такие как синхронизация (MPI_Barrier) и пересылка данных от головного всем процессам (MPI_Bcast). Требуется доступ на запись к некоторой директории на общем файловом сервере (как правило, NFS), видимой одинаковым образом со всех узлов параллельной ВС (кластера).

Таблица 3.3.

Тестируемые функции MPI

Условное название	Описание
TIMING	время на один замер времени
BARRIER	барьерная синхронизация всех процессов
ALLREDUCE	суммирование *N* целочисленных переменных по всем узлам с рассылкой результата по всем узлам
REDUCE	суммирование *N* целочисленных переменных по всем узлам, результат на одном узле
BROADCAST	рассылка *N* целочисленных значений с одного узла по всем узлам
GATHER	сбор *N* целочисленных значений со всех процессов на один головной процесс; в результате головной процесс принимает *N*(*np*-1) целочисленных значений
ALLGATHER	каждый процесс посылает всем остальным процессам *N* целочисленных значений; в результате каждый процесс посылает *N*, а принимает *N*(*np*-1) целочисленных значений; эту операцию можно реализовать как GATHER (сбор), а потом BCAST (рассылка)
ALL-TO-ALL	каждый процесс посылает каждому по *N* целочисленных значений; в результате, каждый процесс посылает и принимает по *N*(*np*-1) целочисленных значений
ISEND & WAIT	неблокирующая посылка *N* целочисленных значений от одного процесса другому с ожиданием завершения
BLOCKING SEND	блокирующая пересылка *N* целочисленных значений от одного процесса другому целочисленных значений
SENDRECV	операция посылки и получения *N* целочисленных значений
SEND & RECV	посылка, а затем получение *N* целочисленных значений
SIGNAL SENDING	посылка сигнала, т. е. сообщения нулевой длины, и получение ответного сигнала (получаемое время должно примерно равняться удвоенной латентности)

Алгоритм тестирования производительности общего доступа к файл-серверу состоит в следующем. На узлах кластера запускается некоторое количество (*np*) MPI-процессов. Каждый отдельный тест проводится в два этапа – запись и чтение. Все процессы синхронизируются до и после окончания каждого этапа.

Для записи и чтения файлов в памяти каждого процесса выделяется буфер определенного размера, причем файлы записываются и читаются порциями, равными размеру буфера. Безусловно, размер буфера влияет на производительность чтения и записи. По умолчанию выбирается размер буфера, равный размеру файла.

В целях проверки перед началом записи каждый из процессов заполняет буфер байтами с номером этого процесса (*myid*). На этапе записи каждый из процессов последовательно записывает на файл-сервер определенное число (*n*) файлов определенного размера (*FS*). Таким образом, на файл-сервере создается $n \times np$ файлов общего размера $TFS = FS \times n \times np$. Измеряется время (*WTT*), затраченное на проведение операций записи (вместе с закрытием) файлов на всех процессорах. Вычисляется общая пропускная способность файл-сервера на тесте записи как $TWT = TFS/WTT$.

На этапе чтения каждый из процессов последовательно прочитывает с файл-сервера *n* ранее записанных файлов размера *FS*. С целью исключить эффект кэширования тест реализован так, что все или большинство из этих файлов будут файлами, созданными другими процессорами. Это условие не будет выполнено, только при работе теста на одном или двух узлах. Измеряется время (*RTT*), затраченное на проведение операций чтения всех файлов на всех процессорах. Вычисляется общая пропускная способность файл-сервера на тесте чтения как $TRT = TFS/RTT$.

Вся процедура тестирования, как правило, состоит из нескольких итераций, в рамках каждой размер файлов (*FS*) изменяется от наименьшего до наибольшего. После окончания всей процедуры выдается наилучшая достигнутая производительность отдельно для операций записи и чтения.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Перечислите основные требования к аппаратному обеспечению кластера.
2. Как влияет время наработки на отказ компьютеров кластера на решение задач с большим временем счета?
3. Как влияет на время решения задачи на кластере объем кэша и быстродействие основной памяти?

4. Что входит в основной комплекс системных программных средств кластера?
5. Какие версии пакета MPICH существуют?
6. Опишите порядок инсталляции пакета MPICH.
7. Для чего предназначена библиотека PETSc.
8. Как инсталлировать библиотеку PETSc?
9. Дайте определение работоспособности и производительности кластера.
10. Как определить работоспособность кластера?
11. От чего зависит производительность кластера?
12. Назовите тесты производительности, разработанные в НИВЦ МГУ.
13. Как определяется время передачи пакета данных?
14. Что такое латентность и как она определяется?
15. Для чего предназначен и как выполняется тест transfer?
16. Как различаются пропускная способность однонаправленных пересылок и пропускная способность двунаправленных пересылок?
17. Для чего предназначен и как выполняется тест nettest?
18. Для чего предназначен и как выполняется тест mpitest?
19. Для чего предназначен и как выполняется тест nfstest?

II. РАБОТА НА КЛАСТЕРЕ

Глава 4. ЗАПУСК И ВЫПОЛНЕНИЕ ПРИЛОЖЕНИЙ НА КЛАСТЕРЕ

4.1. ЭТАПЫ ПОДГОТОВКИ И ВЫПОЛНЕНИЯ ПРИЛОЖЕНИЯ

Подготовка и выполнение параллельной программы включает следующие необходимые этапы.

- Создается проект приложения в соответствии с правилами для среды Visual C++, производятся необходимые настройки, подключаются необходимые библиотеки и заголовочные файлы.
- Создается код программы, производится его компиляция, компоновка и получение исполняемого файла.
- Затем исполняемый файл программы запускается на исполнение из командной строки командой **mpirun** в однопроцессорном варианте (структура команды описана в документации пакета MPICH). Если имеются ошибки запуска, то они исправляются в соответствии с типом ошибки. Если имеются вычислительные ошибки, то для их исправления используется встроенный отладчик Visual C.
- С помощью последовательного отладчика можно исправить только ошибки процесса вычислений, но не коммуникационные. Для исправления коммуникационных ошибок можно использовать метод нескольких последовательных отладчиков (см. гл. 5) или какой-либо другой метод.
- После получения логически корректной программы следует оценить ее эффективность, т. е. определить время ее выполнения, найти ускорение, коэффициент утилизации (использования процессоров) и другие параметры. Это можно сделать, запустив программу на тестовых примерах. При этом тесты должны соответствовать реальным условиям применения программы. Например, не имеет смысла тестировать на эффективность программу умножения матриц размера 100×100, поскольку для таких матриц кластер не нужен. Если параметры эффективности низкие, необходимо с помощью средств профилирования (гл. 6) определить причину этого и найти способ улучшить эффективность. Для повышения эффективности могут применяться самые разные методы: от изменения алгоритма и программы задачи до перестройки программной и аппаратной части кластера. Выбор метода зависит от важности решаемой задачи.

• К эффективности программ относится понятие «масштабирование», сведения о котором приведены в гл. 7.

4.2. ОРГАНИЗАЦИЯ РАБОТЫ ПОЛЬЗОВАТЕЛЯ

Поскольку кластер представляет собой вычислительную сеть из ПЭВМ, было бы удобно, чтобы каждый пользователь имел возможность работать со своими документами и программами независимо от того, за каким компьютером он находится, запуская параллельную программу. Пользователь (или администратор) должен позаботиться о том, чтобы доступ к пользовательской программе был возможен из каждого узла вычислительной сети (иначе программа не сможет запуститься). Пользователи не должны создавать помех друг другу, иметь гарантию, что файлы одного пользователя не будут прочитаны, изменены или удалены другим пользователем.

Сетевые операционные системы позволяют организовать работу пользователя указанным выше образом. Для этого необходимо иметь папки для каждого пользователя, к которым открыт доступ со всех компьютеров кластера, но разрешение на доступ имеет только пользователь – владелец файлов.

Можно выделить два способа запуска параллельных приложений для ОС Windows с использованием пакета MPICH на вычислительном кластере.

1. Если на одном или нескольких компьютерах кластера установлена ОС семейства Server, то запуск параллельной программы для любой конфигурации компьютеров кластера должен осуществляться с такого компьютера. В таком случае для организации работы удобно для каждого пользователя создать сетевой диск, данные которого физически будут находиться на компьютере с ОС Server. Это позволяет осуществлять запуск параллельных программ и работать со своими документами и программами с любого компьютера вычислительной сети. Диск для каждого зарегистрированного пользователя может быть свой – его создает администратор сети.
2. Если ОС Server не установлена либо недоступна, необходимо перед запуском параллельной программы скопировать ее на каждый компьютер, на котором программа должна быть запущена. Локальные адреса программы на разных компьютерах могут быть разными. Для работы кластера удобно на каждом компьютере создать директорию **shared** с общим доступом со всех компьютеров, поддиректо-

рии которого будут личными папками пользователей с настроенными для них правами.

Для Microsoft Windows NT/XP папки пользователей можно подключить как локальные диски командой

net use m: \\host\user

где *m* – имя сетевого диска, *\\host\user* – адрес сетевой папки.

4.3. НАСТРОЙКА КОМПИЛЯТОРА ДЛЯ РАБОТЫ С MPICH

Первое, что нужно для запуска MPI-программы, – откомпилировать программу для используемой реализации MPI, в нашем случае это MPICH. Для этого копируется исходный код в директорию пользователя и запускается компилятор с соответствующими параметрами (либо его запускает среда программирования).

Версии MPICH 1.2.x для NT поставляются со следующими инструкциями по настройке среды Visual C++ 6.0.

1. В настройки компилятора нужно добавить пути к файлам *.h и *.lib, которые используются в проекте. Это можно сделать несколькими способами:
   - Прописав системные переменные

```
set INCLUDE=%mpich%\include;
        %mpich%\mpe\include;%INCLUDE%
set LIB=%mpich%\lib;%LIB%
```

где %mpich% – директория, куда установлен MPICH. Подразумевается, что библиотека MPE установлена в %mpich%\mpe. Эти переменные можно установить, вызвав описанные команды из консоли, с которой будет запускаться компилятор (либо среда разработки).

- Если системная политика позволяет пользователю устанавливать системные переменные для ОС, это можно сделать, выбрав меню «Свойства системы», как это показано в 3.2.1.
- Среда разработки MS VC++ 6.0 позволяет устанавливать директории в настройках окна Options. На закладке Directories нужно добавить пути к INCLUDE и LIB файлам (см. рис. 4.1, 4.2).

2. Затем либо при компиляции из командной строки, либо в свойствах проекта среды MS VC++, нужно указать компилятору параметры:
   - для конфигурации проекта версии Release – установить ключ /MT (use run-time library: multithreaded), добавить компоновщику файлы ws2_32.lib mpich.lib;

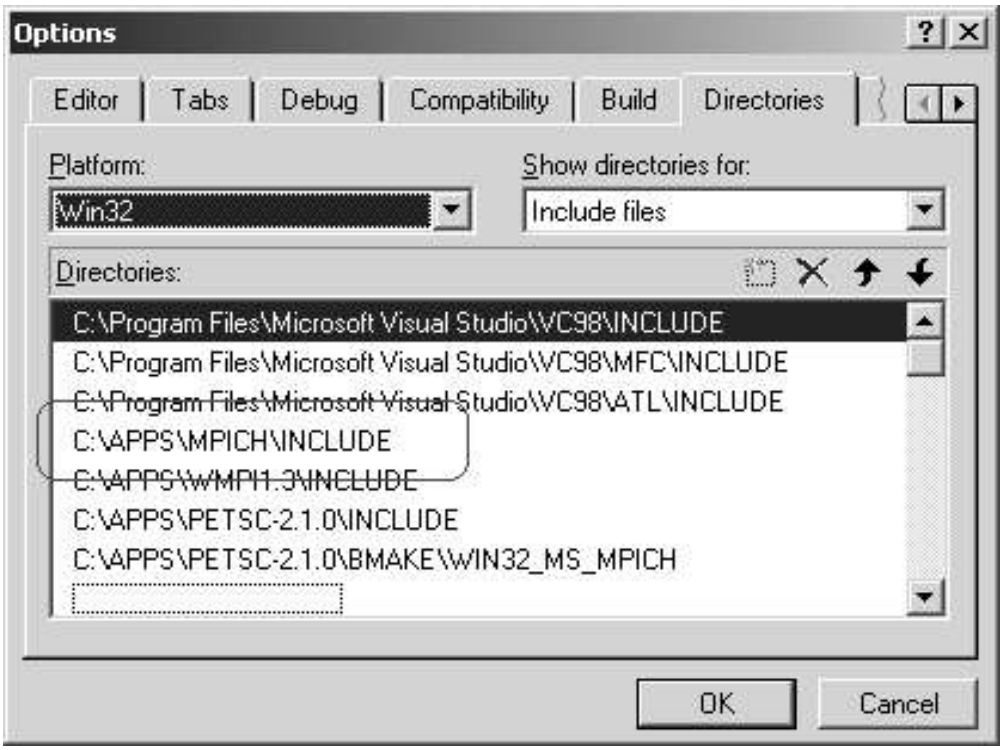


Рис. 4.1. Установка пути для include-файлов

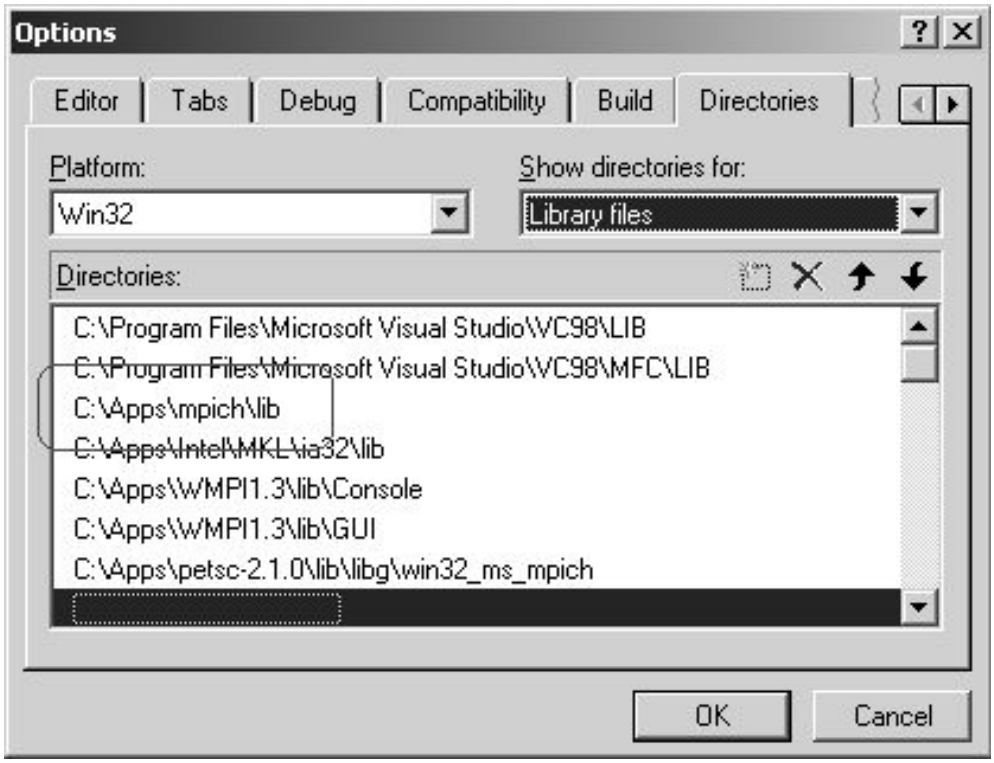


Рис. 4.2. Установка пути для lib-файлов

- для конфигурации проекта версии Debug – установить ключ /MTd (use run-time library: multithreaded), добавить компоновщику файлы ws2_32.lib mpichd.lib.

В VC++ это можно сделать в графическом интерфейсе (см. рис. 4.3–4.5; необходимые действия выделены черным овалом). Нужно обратить внимание на следующее:

- если на машине более одной реализации MPI, то пути к файлам include и lib используемой реализации должны быть прописаны ранее путей к другим реализациям в переменных среды INCLUDE и LIB (или в диалоге Options), иначе программа будет пытаться компилироваться с другой реализацией MPI;
- если при компиляции возникают ошибки типа «unresolved external symbol», нужно убедиться в том, что программа компонуется с библиотеками ws2_32.lib и mpich.lib.

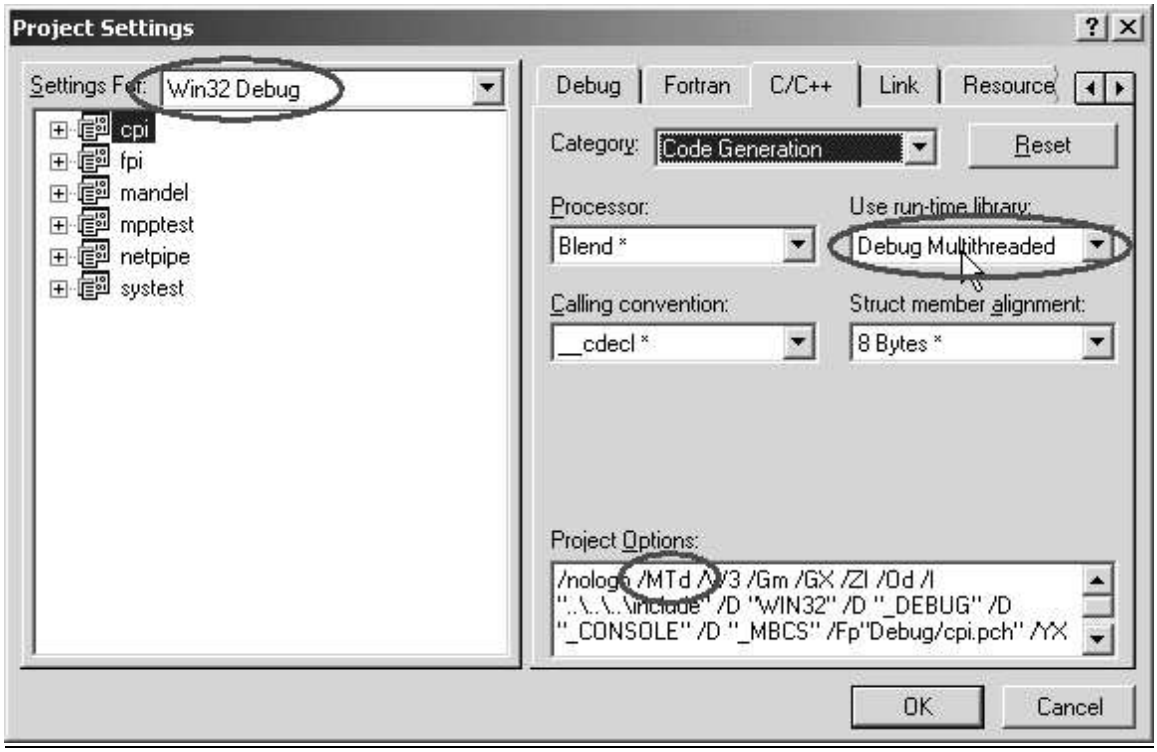


Рис. 4.3. Настройка свойств проекта в VC++ . Установка опции для «use run-time library» Debug-конфигурации

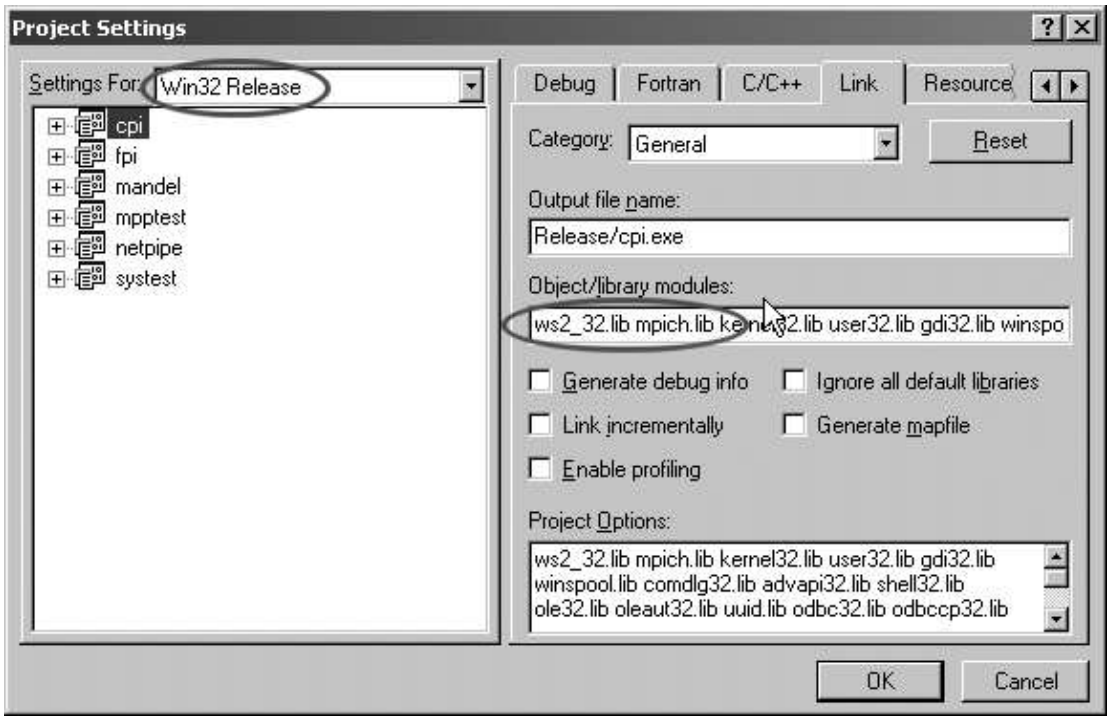


Рис. 4.4. Подключение библиотек MPICH компоновщику для Release-конфигурации в свойствах проекта для VC++

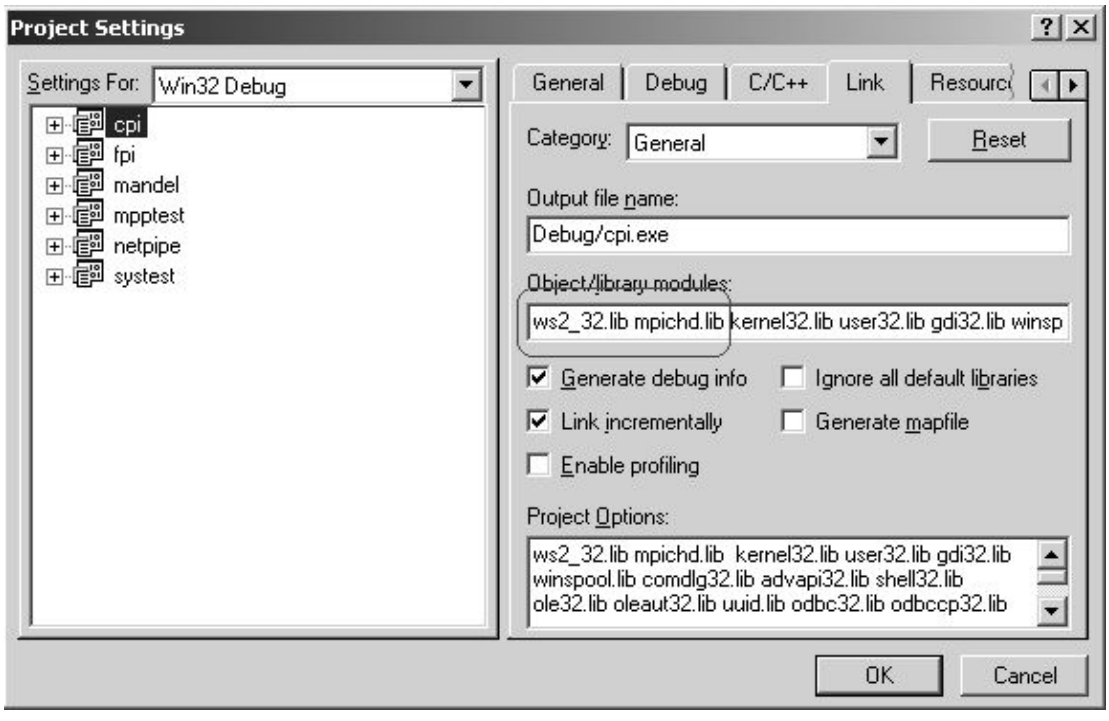


Рис. 4.5. Подключение библиотек MPICH компоновщику для Debug-конфигурации в свойствах проекта для VC++

4.4. ЗАПУСК MPI-ПРОГРАММ НА ОДНОМ ПРОЦЕССОРЕ

Запуск программ в MPICH 1.2.x для Windows NT на одном компьютере осуществляется с помощью команды

mpirun -localonly [num] [program]

где [num] – количество процессов; [program] – имя бинарного файла параллельной программы.

Программа запускается на локальной машине, на которой произведён вызов команды.

Есть возможность запуска программы в локальном режиме с опцией -tcp, при этом программа выполняется локально, но обмен между процессами выполняется через TCP/IP.

Программа, откомпилированная с MPICH, может быть запущена без mpirun, как обычная не MPI-программа из командной строки, что равносильно запуску с одним процессом.

Если сама параллельная программа использует параметры командной строки (например, для чтения входных данных), параметры указываются после имени программы (в конце команды).

Если для программы необходимо задать некоторые переменные, они могут устанавливаться с помощью опции -env. Если для программы необходимы дополнительные аргументы (например, имя обрабатываемого файла), они описываются в конце вызова по стандартным правилам передачи параметров программе:

mpirun -localonly [num] -env "var1=val1|var2=val2|..." [program][args]

Здесь [num] – число процессов; в виде строки описываются переменные, где var* – имя задаваемой системной переменной, val* – значение соответствующей переменной; [args] – аргументы программы.

Однопроцессорный режим может быть полезен для отладки программ с помощью стандартных отладчиков.

4.5. ЗАПУСК ПРИЛОЖЕНИЙ В МНОГОПРОЦЕССОРНОМ РЕЖИМЕ

Запуск в многопроцессорном режиме осуществляется двумя способами:

- mpirun -np [num] [program]
- mpirun file.cfg

В первом случае вызов программы mpirun аналогичен вызову в локальном режиме. Опция -np указывает на многопроцессорный режим, [num] – количество процессов, [program] – имя бинарного файла параллельной программы. Имена компьютеров для выполнения про-

граммы и количество процессов на каждом из них определяются в MPICH.

Если необходимо указать конкретное распределение процессов по процессорам, необходим вызов программы mpirun с файлом конфигурации. Имя файла конфигурации может быть любым, формат файла следующий:

```
exe mpiprogram.exe
[env var1=val1var2=val2var3=val3...]
[dir drive:\my\working\directory]
[map drive:\\host\share]
[args arg1 arg2 ...]
hosts
hostname1 $procs1 [path\mpiprogram.exe]
hostname2 $procs2 [path\mpiprogram.exe]
hostname3 $procs3 [path\mpiprogram.exe]
#...
```

В квадратных скобках указаны необязательные параметры. Комментарии начинаются с символа # и заканчиваются концом строки. После ключевого слова **exe** указывается имя запускаемого файла. За ключевым словом **host** перечисляются сетевые имена узлов (в данном примере hostname*). После каждого имени узла ставится количество процессов ($procs*), запускаемых на этом узле. Для каждого узла оно может быть разным.

Необязательные параметры используются в следующих случаях:

- env – задает некоторые переменные для запускаемой программы;
- args – задает аргументы запускаемой программе;
- map и dir позволяют подключить к файловой системе сетевой диск для программы, установить рабочую директорию.

Для каждой программы пишется свой файл конфигурации. На размещение этого файла в файловой системе ограничений нет, так как его имя указывается при вызове программы запуска mpirun.

В некоторых случаях при запуске приходится использовать следующие утилиты:

- **mpd** – сервис для управления процессами в Windows NT/XP. Фактически производит параллельный запуск программы при вызове команды mpirun. Устанавливается как сервис командой mpd -install, удаляется командой mpd -remove. Может быть установлен в многопользовательском режиме (каждый пользователь должен иметь свои параметры входа на каждом из компьютеров кластера) и в однопользовательском режиме (все программы запускаются с одинаковым регист-

рационным именем, на всех машинах тогда должен быть одно и то же имя). Также mpd может быть вызван в консольном режиме: -console или -console host, где host – имя машины. В консольном режиме предоставляются команды для редактирования настроек кластера: имени пользователя, списка машин, пути временного файла;

- **mpiconfig** – утилита, позволяющая сконфигурировать вычислительную сеть: обнаружить машины в сети, сохранить в реестр список машин, путь для хранения временных файлов, таймаут запуска и др.;
- **mpiregister** – позволяет прописать в реестр имя пользователя, с правами которого запускаются параллельные программы, и его пароль, либо удалить его из реестра, вызвав mpiregister -remove.

4.6. ПРИМЕР ЗАПУСКА И ВЫПОЛНЕНИЯ СТАНДАРТНЫХ ПАРАЛЛЕЛЬНЫХ ПРИЛОЖЕНИЙ

Приводимые примеры конкретизированы для кластера с именем K7, который содержит семь персональных компьютеров и версию MPICH 1.2.5 для Windows NT в исходных кодах.

Для организации работы на компьютере с ОС Microsoft Windows 2000 Server создан сетевой диск *m.* Для работы со своими документами и программами пользователь должен создать на диске *m* папку со своим именем, в которой может хранить файлы и осуществлять запуск параллельных программ.

Пример 1. Приведём инструкцию по запуску стандартного примера – расчёта числа π методом численного интегрирования.

1. Скопируем содержимое *%mpich%\examples\nt\basic*, где расположена данная программа, в папку пользователя, например, в *m:\Sidorov* (либо *C:\Work\MPICH\Sidorov*, если диск *m* отсутствует или не доступен), чтобы иметь возможность скомпилировать ее.
2. Открываем cpi.dsp – файл проекта в VC++.
3. Выполняем соответствующие настройки среды (как описано в параграфе 4.3) в Tools\Options\Directories и Project\Settings При этом нужно проследить, чтобы компилировалась Release-версия, так как в ней присутствует оптимизация по времени исполнения. Компилируем программу с помощью меню Build. Результатом трансляции будет *Release\cpi.exe*.
4. Эту программу можно запустить как обычное консольное приложение, что будет равносильно запуску с одним процессом. Программа запросит количество интервалов, на которое нужно разбить область интегрирования, вычислит и выведет на экран результат

(comp67-1 – имя компьютера в сети, на котором выполняется параллельная программа):

```
Process 0 on comp67-1.
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415927369231227,
Error is 0.0000000833333296
wall clock time = 0.004519
```

5. Программа может быть выполнена на многопроцессорной системе. Для этого запустим ее с помощью команды mpirun. Вызовем из командной строки

   mpirun -localonly 2 cpi.exe

   Программа будет выполняться с двумя процессами на локальной машине. Она будет выполняться медленнее, чем на многопроцессорном компьютере, но такой режим удобен для отладки. Результат может быть такой:

```
Process 1 on comp67-1.
Process 0 on comp67-1.
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415927369231254,
Error is 0.0000000833333322
wall clock time = 0.004612
```

   Если программа mpirun не найдена, нужно указать ее полный путь в *%mpich%\bin*.

6. Теперь запустим программу на кластере. При запуске программы с компьютера с ОС Server (диска *m* кластера К7) выполним команду

   mpirun -np 2 cpi.exe

   При запуске программы с другого компьютера кластера перед вызовом этой команды необходимо программу cpi.exe скопировать на все компьютеры кластера в папки с одинаковыми именами (например, *C:\Work\MPICH*). Результат работы программы для пользователя должен быть таким же, как и при использовании режима -localonly, за исключением того, что запускаемые процессы выведут на экран имена компьютеров (например, comp67-1, comp67-2), на которых они запускаются, и эти имена должны быть разными:

```
Process 1 on comp67-1.
Process 0 on comp67-2.
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415927369231254,
Error is 0.0000000833333322
wall clock time = 0.003487
```

7. Если имена компьютеров кластера одинаковы, это означает, что система не настроена должным образом. Пользователь должен убедиться, что все компьютеры включены. Для этого запустим программу с помощью файла конфигурации. Чтобы узнать имена машин, на которых будет запускаться программа, воспользуемся программой **mpiconfig**, которая позволит определить имена доступных компьютеров кластера. Для этого достаточно набрать имя программы в командной строке. Выбрав режим «Select», можно определить количество машин в кластере и их имена.
8. Создадим в папке с программой cpi.exe файл с названием cpi.cfg (или любым другим). Откроем его с помощью любого текстового редактора, запишем туда следующее:

   ```
   exe cpi.exe
   hosts
   comp67-1 1
   comp67-2 1
   ```

   Затем сохраним файл. Если необходимо, копируем программу cpi.exe на компьютеры comp67-1,comp67-2 кластера в папки с одинаковыми именами (например, *C:\Work\MPICH*). Выполним команду

   mpirun cpi.cfg

   Программа запуститься на компьютерах с сетевыми именами comp67-1 и comp67-2, по одному процессу на каждый (если система правильно настроена и эти компьютеры включены и входят в сеть). Результат работы программы должен быть аналогичен п. 6.
9. Можно добавить все доступные компьютеры кластера в файл конфигурации, поэкспериментировать с количеством процессов на каждом компьютере. Например:

   ```
   exe cpi.exe
   hosts
   comp67-1 1
   comp67-2 2
   comp67-3 3
   comp67-4 4
   ```

10. В заключение желательно снять график зависимости ускорения от числа компьютеров.

Пример 2. Более сложный пример параллельной программы, которая находится в папке *%mpich%\examples\nt\mandel* и вычисляет множество Мандельброта. Пример демонстрирует запуск параллельной MPI-программы с передачей аргументов ей из командной строки.

1. Скопируем программу в папку пользователя и откомпилируем, как это описано в предыдущем примере.
2. Теперь запустим программу. Особенностью программы mandel является то, что она не может работать на одном процессе. Войдя в папку с исполняемым файлом, наберем в командной строке:

   mpirun -np 2 mandel.exe

   Если система настроена правильно, запустится пример, вызывающий последовательность изображений на экране.
3. Можно запустить программу с помощью файла конфигурации. Создадим файл с любым именем (например, mandel.cfg), с помощью текстового редактора запишем в файле:

   ```
   exe mandel.exe
   hosts
   comp67-1 1
   comp67-2 1
   ```

   Запуск программы с помощью такого файла конфигурации означает, что программа mandel.exe параллельно выполняется на двух компьютерах, имена которых comp67-1 и comp67-2. На экран компьютера comp67-1 будет выведено окно с результатом вычислений.
4. Программа mandel может быть запущена с аргументами. Список аргументов программа выдает при запуске ее с одним процессом. Примеры аргументов представлены в файле cool.points, который находится в исходной папке, кроме того, сам этот файл может быть аргументом программы. Вызов программы в этом случае производится из командной строки следующим образом:

   mpirun -np 2 mandel.exe -i cool.points

   или, через файл конфигурации:

   mpirun mandel.cfg

   Файл конфигурации следует изменить следующим образом:

   ```
   exe mandel.exe
   args -i cool.points
   hosts
   comp67-1 1
   comp67-2 1
   ```

   На экран будет выведена последовательность изменяющихся цветных изображений, параметры которых заданы в файле cool.points.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите этапы подготовки и выполнения параллельного приложения.
2. Какие типы ошибок возможны в параллельной программе?
3. Что такое эффективность и масштабирование параллельного приложения?
4. Как организовать работу пользователя для запуска параллельных приложений на кластере?
5. Где располагаются файлы пользователя при запуске параллельных приложеий с помощью пакета MPICH?
6. Назовите основные этапы настройки компилятора для работы с пакетом MPICH.
7. Как запускается параллельная программа в однопроцессорном режиме? Назовите два варианта.
8. Можно ли запустить параллельную программу в однопроцессорном режиме непосредственно из среды программирования?
9. Можно ли запустить параллельную программу в многопроцессорном режиме непосредственно из среды программирования?
10. Какие способы запуска в многопроцессорном режиме Вы знаете?
11. Как запускаются параллельные приложения командой mpirun с опцией –np?
12. Как запускаются параллельные приложения командой mpirun с помощью файла конфигурации?
13. Опишите структуру файла конфигурации.
14. Приведите пример файла конфигурации для запуска программы test.exe, расположенной в папке *C:\Work\Mpich* компьютера с именем comp67-1, с четырьмя процессами:

- два процесса на машине с именем comp67-1, два процесса на машине с именем comp67-2;
- один процесс на машине comp67-1, один процесс на машине comp67-2, два процесса на машине comp67-3;
- один процесс на машине comp67-1, один процесс на машине comp67-2, один процесс на машине comp67-3, один процесс на машине comp67-4.

Глава 5. ОТЛАДКА ПАРАЛЛЕЛЬНЫХ ПРИЛОЖЕНИЙ

5.1. ОСОБЕННОСТИ ОТЛАДКИ ПАРАЛЛЕЛЬНЫХ ПРИЛОЖЕНИЙ

Средства отладки являются необходимой принадлежностью любой системы программирования. Отладчик должен позволять:

- запустить программу;
- остановить программу в заданной точке;
- обеспечить пошаговый просмотр программы;
- просмотреть значения нужных переменных;
- изменить некоторые части программы.

Отладка параллельной программы – процесс более трудоемкий, чем отладка последовательной программы. Это происходит не только из-за сложности параллельной программы, но и из-за ее недетерминированного поведения. Вероятность появления ошибок при написании параллельных программ возрастает, поскольку в задаче необходимо найти подзадачи, способные выполняться одновременно, а также обеспечить коммуникации между ними.

Кроме того, появляется новый класс ошибок – deadlocks. **Deadlock** (тупик) – ситуация, когда, например, первый процесс обращается ко второму, однако это обращение не может быть выполнено, поскольку второй процесс занят, а занят он обращением к первому процессу. Оба процесса находятся в состоянии ожидания, дальнейшее выполнение программы невозможно.

Большинство современных средств отладки параллельных программ основано на представлении программы как совокупности выполняющихся процессов. Задача отладки параллельных программ в большинстве случаев сводится к отладке каждого процесса в отдельности плюс изучение поведения процессов при их взаимодействии. При этом отдельный процесс отлаживается стандартными для последовательных программ средствами, а исследование взаимодействий требует других средств.

Можно выделить три этапа при отладке параллельной программы.

- На первом этапе программа отлаживается на рабочей станции как последовательная программа, с использованием обычных методов и средств отладки.
- На втором этапе программа выполняется на той же рабочей станции в специальном режиме моделирования параллельного выполнения для проверки корректности распараллеливающих указаний.
- На третьем этапе программа может быть выполнена на параллельной машине в специальном режиме – промежуточные результаты параллельного выполнения сравниваются с эталонными результатами (например, результатами последовательного выполнения).

Можно выделить несколько возможностей отладки параллельных приложений:

- трассировка отлаживаемой программы;
- использование последовательных отладчиков;
- использование параллельных отладчиков.

5.2. ТРАССИРОВКА

Трасса – журнал событий, произошедших во время исполнения программы. *Трассировка* – широко используемый метод отладки как при последовательном, так и параллельном программировании. Для трассировки в текст программы вставляются операторы вывода на экран с текстом описания событий.

Для параллельных программ необходимо указать в выводимой строке номер процесса. В C/C++ вывод сообщений можно производить, например, функцией

printf("process %d: state1", myid);

где *myid* – номер процесса, выполняющего сообщение.

Однако следует отметить, что функции ввода-вывода могут замещаться реализациями MPI, как это сделано в MPICH, поэтому трассировка может вызвать проявление одних ошибок и сокрытие других. Иными словами, трассировка не является универсальным средством отладки, хотя может быть полезна в большинстве случаев, особенно когда параллельные отладчики недоступны.

В программе для вычисления значения π это может быть, например, сделано так:

```
#include "mpi.h"
#include <math.h>
int main ( int argc, char *argv[ ] )
{ int n, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264 3;
   double mypi, pi, h, sum, x;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   printf ("begin process %d /n",myid);
   while (1)
   {    if (myid == 0)
        {    printf ("Enter n - the number of intervals: (0 quits) ");
             scanf ("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf (" process %d recieve n /n",myid);
        if (n == 0)  break;
        else
        {   h  = 1.0/ (double) n;
            sum = 0.0;
```

```
            for (i = myid +1; i <= n; i+= numprocs)
            {       x = h * ( (double)i – 0.5);
                    sum += (4.0 / (1.0 + x*x));
             }
             mypi = h * sum;
             printf (" mypi in  process %d = %f /n",myid,mypi);
             MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                              MPI_COMM_WORLD);
              if (myid == 0)
                   printf ("pi is approximately %.16f.”,pi );
        }
    }
    MPI_Finalize();
    return 0;
}
```

Каждый процесс отмечает начало работы, получение переменной *n* и вычисление частичной суммы. Эти операции в программе выделены жирным шрифтом. В результате будет получена трассировка программы, анализ которой позволит найти ошибку.

Если выполнить данную программу с четырьмя процессами трасса событий должна быть следующей:

```
begin process 0
begin process 2
begin process 3
begin process 1
Enter n - the number of intervals: (0 quits) 100
process  2 recieve n
process  1 recieve n
process  0 recieve n
process  3 recieve n
mypi in  process 0 = 0.78539818423078136
mypi in  process 1 = 0.78539818423078135
mypi in  process 2 = 0.78539818423078134
mypi in  process 3 = 0.78539818423078135
pi is approximately 3.141592739231254.
Enter n - the number of intervals: (0 quits) 0
process  2 recieve n
process  1 recieve n
process  0 recieve n
process  3 recieve n
```

Если же трасса событий неполная, то по характеру трассы можно определить участок программы, в котором находится ошибка.

5.3. ОТЛАДКА С ПОМОЩЬЮ ПОСЛЕДОВАТЕЛЬНЫХ ОТЛАДЧИКОВ

Как отмечалось в параграфе 4.4, программа, откомпилированная с MPICH, может быть запущена как обычная программа из командной строки, что равносильно запуску с одним процессом. В таком случае для отладки вычислительной части программы можно использовать все возможности встроенного последовательного отладчика.

Однако последовательные отладчики можно использовать и для анализа коммуникационных ошибок, т. е. ошибок, возникающих при параллельном исполнении процессов. Для этого все или несколько отлаживаемых процессов параллельной программы должны быть запущены под последовательными отладчиками, но со своими значениями некоторых системных переменных. В этом случае программа запускается не с помощью программы mpirun, а «вручную».

Для каждого MPICH-процесса нужно установить системные переменные (такие как количество процессов приложения, номер запускаемого процесса и др.), затем выполнять процессы под последовательными отладчиками.

Если мы запускаем MS Visual C++, то вызов будет следующим:

```
set MPICH_JOBID=$jobhost.$N
set MPICH_IPROC=$myid
set MPICH_NPROC=$numprocs
set MPICH_ROOT=$host:$port
msdev.exe $project
```

Здесь:

- **$jobhost –** имя машины, используемой в качестве сервера задач;
- **$N** – короткая уникальная строка, позволяющая идентифицировать данное задание;
- **$myid** – номер запускаемого процесса;
- **$numprocs** –общее количество процессов;
- **$host** – имя машины, на которой производится отладка;
- **$port** – номер порта, через который процессы соединяются;
- **$project** – имя отлаживаемого проекта.

Установка MPICH_JOBID обязательна в случае, если MPICH настроен на использование сервера задач. Определить или изменить эту настройку можно, вызвав программу *mpiconfig*.

Каждая такая последовательность команд вызывает один процесс. Так как программа запускается на нескольких процессах (количество процессов равно $numprocs), то такая последовательность должна

быть повторена в $numprocs консолях, при этом параметр $myid должен пробегать все значения от 0 до $numprocs – 1, причем значение параметра $numprocs должно быть одинаковым во всех вызовах.

Итак, нужно создать $numprocs bat-файлов, в каждый из которых поместить последовательность вышеописанных команд, но значение параметра $myid в каждом файле будет разным, изменяясь от 0 до $numprocs-1. Описанный метод позволяет отлаживать параллельные программы в исходных кодах.

Например, при отладке программы cpi.dsw на двух процессах следует создать два bat-файла со следующим содержанием.

```
rem ---- start1.bat ----
set MPICH_JOBID=comp67-1.1
set MPICH_IPROC=0
set MPICH_NPROC=2
set MPICH_ROOT=comp67-1:12345
msdev cpi.dsw
```

```
rem ---- start2.bat ----
set MPICH_JOBID=comp67-1.1
set MPICH_IPROC=1
set MPICH_NPROC=2
set MPICH_ROOT=comp67-1:12345
msdev cpi.dsw
```

Из командной строки запускаем файл start1.bat. В результате его работы запустится Visual C++ с программой cpi и номером процесса, равным 0. Аналогично из другой командной строки запускаем файл start2.bat. Запускается еще раз Visual C++ с программой cpi и номером процесса, равным 1.

Результатом запуска обоих файлов будут два окна в VC++, отлаживаемый в каждом окне процесс будет обладать свойствами MPICH-процесса. Затем вызываем в обоих окнах отладчик
Visual C++. Переключаясь между окнами, можно шаг за шагом выполнять оба процесса. При этом имеем все необходимые возможности отладки: выполнение по шагам, установку точек прерываний, просмотр необходимых переменных и т. д.

5.4. ПАРАЛЛЕЛЬНЫЙ ОТЛАДЧИК TOTALVIEW

Для отладки параллельных программ созданы специальные параллельные отладчики. Наиболее известный из них – высокоуровневый

отладчик оконного типа TotalView (TV), который работает под ОС Unix. Пользователи имеют все стандартные возможности современных отладчиков плюс многие дополнительные возможности, спроектированные специально для ускоренной отладки многопроцессных приложений.

Коммерческий отладчик TotalView является переносимым отладчиком для параллельных программ [1]. TV поддерживает множество реализаций MPI, включая MPICH. С помощью TV разработчики способны быстро и точно отладить приложения со многими процессами, потоками и процессорами.

TV имеет следующие возможности:

- процессы могут быть легко запущены, остановлены, перезапущены, просмотрены и удалены;
- пользователь может легко подключиться к любому процессу в системе простым щелчком мыши. Точки останова легко управляются и изменяются;
- программа может быть исправлена «на лету», поэтому различные сценарии исполнения проверяются быстро и без перекомпиляции;
- множественные процессы на множественных процессорах могут быть сгруппированы, и, когда один процесс достигает точки останова, все сгруппированные процессы будут остановлены; это может существенно помочь при отладке распределенных систем, построенных по принципу клиент-сервер;
- распределенная архитектура TV позволяет отлаживать удаленные программы на всей сети.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите основные функции отладчика.
2. Что такое deadlocks?
3. Назовите три этапа отладки параллельной программы.
4. Что такое трассировка?
5. Приведите пример трассы событий для программы вычисления значения π при запуске программы с двумя процессами.
6. Как осуществляется отладка вычислительных ошибок с помощью последовательного отладчика?
7. Как можно использовать последовательные отладчики для определения коммуникационных ошибок?
8. Приведите пример bat-файлов для отладки программы вычисления значения π с четырьмя процессами для Вашего кластера.
9. Определите характер ошибки программы вычисления значения π, если она выполняется с двумя процессами, а трасса событий следующая:

```
begin process 0
Enter n - the number of intervals: (0 quits) 100
process  0 recieve n
```

10. Определите характер ошибки программы вычисления значения π, если она выполняется с двумя процессами, а трасса событий следующая:

```
begin process 0
begin process 1
Enter n - the number of intervals: (0 quits) 100
process  0 recieve n
```

11. Определите характер ошибки программы вычисления значения π, если она выполняется с двумя процессами, а трасса событий следующая:

```
begin process 0
begin process 1
Enter n - the number of intervals: (0 quits) 100
process  1 recieve n
process  0 recieve n
mypi in  process 0 = 1.57079636846155
```

12. Определите характер ошибки программы вычисления значения π, если она выполняется с двумя процессами, а трасса событий следующая:

```
begin process 0
begin process 1
Enter n - the number of intervals: (0 quits) 100
process  1 recieve n
process  0 recieve n
mypi in  process 0 = 1.57079636846155
mypi in  process 1 = 1.57079636846155
Enter n - the number of intervals: (0 quits) 0
process  0 recieve n
```

13. Каковы особенности отладчика TotalView?

Глава 6. ПРОФИЛИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ ПРИЛОЖЕНИЙ

6.1. ПРОФИЛИРОВАНИЕ

При разработке параллельных программ основной целью распараллеливания является увеличение быстродействия, поэтому процесс разработки также должен быть направлен на изучение эффективности вычислений. Было бы полезно знать, какова была последовательность событий при выполнении программы, сколько времени было потрачено на каждую стадию вычисления, сколько времени занимает каждая отдельная коммуникационная операция. Все это позволит выявить и

устранить «узкие места», тем самым уменьшить время выполнения программы. Путь решения проблемы анализа – сохранение последовательности событий и представление этой информации в удобном для программиста виде. Следовательно, нужны файлы событий с привязанными к этим событиям временными отметками и их графическое представление. Такие файлы называются логфайлами, а метод исследования эффективности – профилированием.

6.2. БИБЛИОТЕКА MPE

Для стандарта MPI создана библиотека профилирования **MPE** (**M**ulti-**P**rocessing **E**nvironment). MPE обеспечивает разные способы генерирования логфайлов, которые могут быть просмотрены одним из графических инструментальных средств, распространяемых вместе с MPE. Кроме того, есть возможность добавлять в логфайлы специфическую для приложения информацию.

Библиотека MPE распространяется наряду с эталонной версией MPI и построена таким образом, чтобы она могла работать с любой MPI реализацией. Библиотека MPE полностью вложена в MPICH, что упрощает ее использование.

Чтобы запустить в программе регистрацию событий, необходимо чтобы каждый процесс перед регистрацией вызвал процедуру **MPE_Init_log**. По окончании регистрации каждый процесс должен вызвать **MPI_Finish_log** для объединения информации о каждом процессе, хранящуюся во временных файлах, в один логфайл, указываемый в параметре функции. **MPE_Stop_log** используется, чтобы приостановить регистрацию, хотя таймер продолжает работать. Процедура **MPE_Start_log** возобновляет регистрацию.

Для определения типов событий при регистрации используются любые неотрицательные целые числа. Чтобы получить уникальные номера событий из системы MPE, должна использоваться функция **MPE_Log_get_event_number**.

События рассматриваются как не имеющие продолжительности. Чтобы измерить продолжительность некоторого состояния программы, два события определяются как начало и окончание состояния. Состояние описывается процедурой **MPE_Describe_state**, которая принимает события начала и окончания этого состояния. Для увеличения наглядности процедура MPE_Describe_state также добавляет название события и цвет, которым состояние будет обозначено при визуализа-

ции. Соответствующая процедура **MPE_Describe_event** обеспечивает описание типа для события.

Функции MPE_Describe_event и MPE_Describe_state вызываются только в главном процессе, а функция MPE_Log_event – в каждом процессе, где нужна фиксация зарегистрированного сообщения в MPE_Describe_event или MPE_Describe_state.

Профилирование всегда увеличивает время исполнения программы. Для точности измерений важно, чтобы регистрация события была быстрой операцией. Операция MPE_Log_event сохраняет малое количество информации в памяти, и это требует минимального времени. При выполнении MPE_Finish_log эти буфера параллельно объединяются в конечный буфер, отсортированный по временным меткам, который записывается в файл процессом 0.

6.3. САМОПРОФИЛИРОВАНИЕ

В MPE созданы также библиотеки профилирования, не требующие явной модификации кода программы. Регистрация осуществляется путем замещения вызовов функций MPI на соответствующие профилирующие функции. Самый простой способ генерировать логфайлы состоит в том, чтобы откомпилировать программу со специальной библиотекой MPE, которая использует возможность профилирования MPI перехватом всех вызовов MPI.

Первая библиотека профилирования проста. Версия каждой MPI_Xxx функции вызывает операцию **PMPI_WTIME**, которая поставляет временную метку до и после каждого обращения к соответствующей PMPI_Xxx подпрограмме. Полученные времена для каждого процесса записываются в файлы, один файл на процесс. Эта версия не принимает во внимание вложенные обращения, которые происходят, когда MPI_BCAST, например, реализована в терминах MPI_SEND и MPI_RECV.

Вторая профилирующая библиотека генерирует логфайлы с отсортированными по времени событиями или состояниями. Во время выполнения программы вызовы **MPE_Log_event** сохраняют события некоторых типов в памяти. Эти данные объединяются при выполнении **MPI_Finalize**. Функция **MPI_Pcontrol** в течение выполнения программы может использоваться, чтобы приостановить и перезапустить операции регистрации. По умолчанию, регистрация включена. Вызов **MPI_Pcontrol(0)** выключает регистрацию, **MPI_Pcontrol(1)** включает регистрацию.

Обращения к MPE_Log_event при самопрофилировании происходят автоматически при каждом обращении к MPI. В дополнение к использованию предопределенных библиотек регистраций MPE вызовы регистрации MPE могут быть также вставлены в программу MPI самим пользователем. Эти состояния называются определяемыми пользователем состояниями.

Третья библиотека делает возможной простую форму анимации программы в реальном масштабе времени. Библиотека графики MPE содержит подпрограммы, которые позволяют нескольким процессам совместно использовать дисплей X, который не связан с любым специфическим процессом. Эта позволяет осуществить прорисовку стрелок, которые представляют движение сообщения во время выполнения программы.

В профилирующей библиотеке, перехватывающей вызов некоторой функций MPI, необходимо выполнить ряд действий до и после вызова этой функции. Тело некоторой функции MPI_Xxx будет выглядеть примерно так:

```
int MPI_Xxx( . . . )
{  do something for profiling library
   int retcode = PMPI_Xxx( . . . );
   do something else for profiling library
   return retcode;
}
```

Библиотеки регистрации (Logging Libraries) – наиболее полезные и широко используемые библиотеки профилирования в MPE. Они позволяют сочетать гибкость и быстроту создания параллельных программных продуктов. Регистрационные файлы могут быть созданы вручную, путем вставки вызовов MPE в MPI программу, или автоматически, компонуясь с соответствующими библиотеками MPE, или объединяя вышеупомянутые два метода.

6.4. ФОРМАТЫ ЛОГФАЙЛОВ

В настоящее время в библиотеке MPE используются три различных формата логфайлов. Заданный по умолчанию формат – **CLOG** представляет собой простое перечисление событий в порядке их наступления. Формат **ALOG** поддерживается только для совместимости вниз и в настоящее время не применяется и не разрабатывается. Программа для чтения и представления логфайлов формата ALOG называется upshot. Наиболее мощный формат – **SLOG** (**S**calable **LOG**file

format), масштабируемый формат логфайла. Масштабируемость исходит из разделения всех состояний в логфайле на фреймы данных небольшого размера, что позволяет их эффективно обрабатывать программой визуализации Jumpshot-3. По утверждению разработчиков, SLOG и Jumpshot-3 способны обработать логфайл размером в Гбайт. Следует отметить, что сам формат требует большего дискового пространства, чем ALOG и CLOG.

Набор утилит MPE включает преобразователь форматов (например, clog2slog, clog2alog), средства печати логфайлов и просмотрщик (logviewer) логфайлов, который выбирает правильное графическое средство, чтобы отобразить логфайл, основанный на расширении логфайла. В настоящее время графические инструментальные средства MPE включают три программы просмотра логфайлов: upshot для ALOG, Jumpshot-2 для CLOG и Jumpshot-3 для SLOG.

Файл SLOG формата может быть получен из CLOG при помощи программы clog2slog.exe или непосредственным генерированием во время исполнения MPI-программы. Для такого генерирования нужно присвоить переменной среды MPE_LOG_FORMAT значение SLOG.

Существуют две переменные, читаемые библиотекой MPE, которые пользователь может установить перед порождением логфайлов:

- MPE_LOG_FORMAT – определяет формат логфайла, сгенерированного при выполнении приложения, связанного с библиотеками регистрации MPE. Возможные значения для MPE_LOG_FORMAT: CLOG, SLOG и ALOG. Когда значение формата не установлено в MPE_LOG_FORMAT, по умолчанию принят CLOG;
- TMPDIR – определяет директорию, которую нужно использовать для хранения временной информации каждого процесса. По умолчанию, когда TMPDIR не установлен, используется директория /tmp. Когда пользователь вынужден генерировать очень большой логфайл для долго выполняющейся задачи, он должен удостовериться, что в TMPDIR достаточно места, чтобы вместить временный логфайл, который будет удален, если объединенный логфайл будет создан успешно. Чтобы минимизировать непроизводительные затраты при регистрации, рекомендуется использовать локальную файловую систему для TMPDIR. Объединенный логфайл будет записан в файловую систему, где расположен процесс с номером 0.

Библиотека MPE построена в предположении, что часы на узлах распределенной системы синхронизированы, но это не всегда выполняется. В сетях автоматизированных рабочих мест ситуация намного

хуже, и время иногда дрейфует в одном узле относительно другого. Это означает, что перед профилированием необходимо произвести синхронизацию часов, иначе измерения будут несостоятельными.

Среда MPE обеспечивает программистов полным набором инструментальных средств анализа эффективности MPI-программ. Эти инструментальные средства включают набор библиотек профилирования, утилит и графических инструментальных средств. Библиотека профилирования MPE постоянно развивается.

6.5. ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ ПРОФИЛИРОВАНИЯ MPE

Сведения конкретизированы для версии MPICH NT1.2.x. Не все реализации MPI и даже не все версии MPICH содержат библиотеку MPE. Однако библиотека MPE совместима с любой реализацией MPICH, но может иметь нюансы при использовании с разными реализациями.

Чтобы использовать MPE, сначала откомпилируем профилирующуюся программу. Для этого можно воспользоваться самопрофилирующей библиотекой – все вызовы MPI будут сохранены в логфайле.

В ОС Windows NT/XP при использовании Visual C++ нужно поместить в список компоновки библиотеку mpe.lib перед mpich.lib. В этом случае будут использоваться самопрофилирующие варианты вызовов функций MPI из библиотеки MPE. Если определен конфликт mpe.lib со стандартной библиотекой LIBCMT.lib, компоновщику нужно добавить опцию /NODEFAULTLIB:"LIBCMT.lib".

Откомпилированная с такими опциями программа создает логфайл, в котором запоминается время всех вызовов функций MPI.

6.6. ВИЗУАЛИЗАЦИЯ РЕЗУЛЬТАТОВ

Для визуализации результатов профилирования в MPICH имеются программы jumpshot-2 и jumpshot-3. Эти программы реализованы на языке java и требуют версию JVM не ранее 1.2. Запуск jumpshot-3 может производиться, например, следующей командой:

java -jar jumpshot3.jar

Если запуск производится не в той директории, где находится файл jumpshot3.jar, должен быть указан полный путь. Для Windows можно создать bat-файл, который будет запускать визуализацию логфайла при вызове

nupshot logfile.slog

где logfile.slog – логфайл в формате SLOG, получившийся в результате исполнения программы, либо после выполнения clog2slog над логфайлом в формате CLOG, либо другой соответствующей утилиты.

Вызов jumpshot-3 в этом случае из файла будет выглядеть так:

java -jar %путь%\jumpshot3.jar %1

Логфайлы позволяют проанализировать ход выполнения параллельной программы, возникающие задержки из-за рассинхронизации процессов, что позволит программисту создавать более эффективные программы.

6.7. ПРИМЕР ПРОФИЛИРОВАНИЯ

В качестве примера профилирования выберем стандартный пример профилирования программы **cpilog** – программы вычисления числа π с профилированием. Программа выглядит следующим образом:

```
#include "mpi.h"
#include "mpe.h"
#include <math.h>
#include <stdio.h>
double f(double a)
{    return (4.0 / (1.0 + a*a));  }

  int main(int argc, char *argv[])
{ int  n, myid, numprocs, i, j;
 double PI25DT = 3.14159265358979323846264 3;
 double mypi, pi, h, sum, x,startwtime = 0.0, endwtime;
 int namelen, event1a, event1b, event2a, event2b, event3a, event3b, event4a, event4b;
 char pr_name[MPI_MAX_PROCESSOR_NAME];

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 MPI_Get_processor_name(processor_name,&namelen);
 fprintf(stderr,"Process %d running on %s\n", myid, pr_name);

  MPE_Init_log();      /* Get event ID from MPE, user should NOT assign event ID*/
  event1a = MPE_Log_get_event_number();
  event1b = MPE_Log_get_event_number();
  event2a = MPE_Log_get_event_number();
  event2b = MPE_Log_get_event_number();
  event3a = MPE_Log_get_event_number();
  event3b = MPE_Log_get_event_number();
  event4a = MPE_Log_get_event_number();
  event4b = MPE_Log_get_event_number();
```

```
    if (myid == 0)
    {   MPE_Describe_state(event1a, event1b, "Broadcast","red");
        MPE_Describe_state(event2a, event2b, "Compute","blue");
        MPE_Describe_state(event3a, event3b, "Reduce","green");
        MPE_Describe_state(event4a, event4b, "Sync", "orange");
    }
    if (myid == 0)
    {   n = 1000000;
       startwtime = MPI_Wtime();
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPE_Start_log();
    for (j = 0; j < 5; j++)
    {     MPE_Log_event(event1a, 0, "start broadcast");
          MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
          MPE_Log_event(event1b, 0, "end broadcast");
          MPE_Log_event(event4a,0,"Start Sync");
          MPI_Barrier(MPI_COMM_WORLD);
          MPE_Log_event(event4b,0,"End Sync");
          MPE_Log_event(event2a, 0, "start compute");
          h   = 1.0 / (double) n;
          sum = 0.0;
          for (i = myid + 1; i <= n; i += numprocs)
          {   x = h * ((double)i - 0.5);
             sum += f(x);
          }
          mypi = h * sum;
          MPE_Log_event(event2b, 0, "end compute");
          MPE_Log_event(event3a, 0, "start reduce");
          MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                             MPI_COMM_WORLD);
          MPE_Log_event(event3b, 0, "end reduce");
    }
     MPE_Finish_log("cpilog");
     if (myid == 0)
     {     endwtime = MPI_Wtime();
           printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
           printf("wall clock time = %f\n", endwtime-startwtime);
      }
    MPI_Finalize();
    return(0);
}
```

Откроем в Visual C++ исходный код программы, размещенный в *%mpich%\examples\basic\cpilog.c*. Установим конфигурацию Release, как это описано в параграфе 4.6, откомпилируем программу. Компилятор должен выдать ряд строк.

```
Compiling...
cpilog.c
Linking...
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Finalize
cpilog.obj : error LNK2001: unresolved external symbol _MPE_Finish_log
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Reduce
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Bcast
cpilog.obj : error LNK2001: unresolved external symbol _MPE_Log_event
cpilog.obj : error LNK2001: unresolved external symbol _MPE_Start_log
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Barrier
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Wtime
cpilog.obj : error LNK2001: unresolved external symbol _MPE_Describe_state
cpilog.obj : error LNK2001: unresolved external symbol
                                                  _MPE_Log_get_event_number
cpilog.obj : error LNK2001: unresolved external symbol _MPE_Init_log
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Get_processor_name
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Comm_rank
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Comm_size
cpilog.obj : error LNK2001: unresolved external symbol _MPI_Init
Debug/cpilog.exe : fatal error LNK1120: 15 unresolved externals
Error executing link.exe.
```

Если ошибки возникли на этапе компиляции, следует настроить компилятор, как описано в параграфе 4.3. Причина таких ошибок – не подключена библиотека с реализацией функций (или с реализацией вызова функций из dll – динамически подключаемой библиотеки). Откроем свойства проекта, добавим компоновщику файлы mpe.lib, mpich.lib, добавим в параметры /nodefaultlib библиотеку libcmt.lib, ещё раз запустим компиляцию, результатом которой должен стать файл Release\cpilog.exe.

Программа откомпилирована, но это не значит, что она работает корректно. Обратим внимание на сообщение:

```
/* MPE_Init_log & MPE_Finish_log are NOT needed when  liblmpe.a is linked
   with  this program.  In that case,  MPI_Init would have called MPE_Init_log
   already.
*/
```

Так как компилировали программу с самопрофилирующей библиотекой, то функции MPE_Init_log() и MPE_Finish_log() вызываются

автоматически, поэтому повторные вызовы в программе следует удалить. Теперь запустим полученную программу с помощью команды mpirun, как показано в примере параграфа 4.6. При завершении программы появляется файл с названием cpilog.exe.clog. Конвертируем этот файл в формат SLOG командой

clog2slog cpilog.clog

Затем запустим средство визуализации jumpshot-3:

java -jar %путь%\jumpshot3.jar cpilog.exe.slog

Для этого в системе должна быть установлена java-машина, существовать файл jumpshot3.jar, который в последних версиях поставляется в исходном коде и под Windows не компилируется без специальных настроек.

Путь к java-архиву для jumpshot3.jar должен быть указан полностью. В целях упрощения процесса визуализации создан bat-файл upshot.bat для запуска визуализации. Вместо длинного вызова java-программы достаточно ввести команду

upshot cpilog.exe.slog

Нажав на кнопку «Display», мы получим рис. 6.1, на котором каждая горизонтальная полоса представляет один процесс программы.

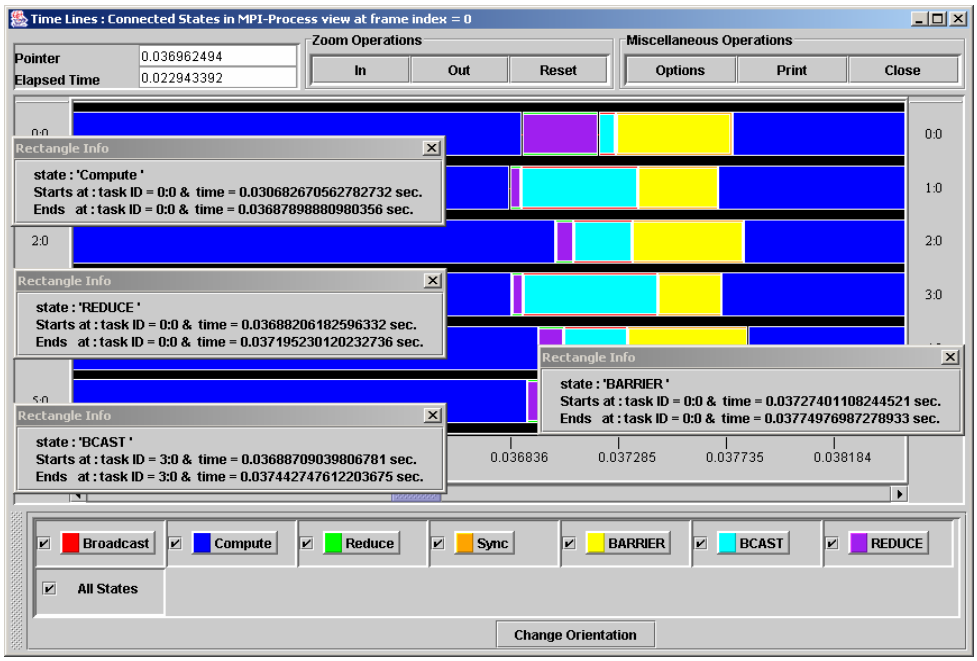


Рис. 6.1. Визуальное представление логфайла программы cpilog

Разным цветом обозначены различные состояния программы. Поскольку смена состояний всех процессов привязана к единой временной шкале, то, изучая диаграммы, можно установить время выполнения различных участков любого процесса, в частности оценить время операций обмена и время вычислений. Это позволяет выявить «узкие» места программы. Кроме того, диаграммы позволяют выявить взаимные задержки процессов.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое библиотека MPE, ее назначение?
2. Основные функции MPE?
3. Что такое профилирование?
4. Что такое самопрофилирование?
5. Какова методика оценки эффективности вычислений?
6. Что такое регистрация, логфайлы?
7. Как создается файл регистрации?
8. Какие процедуры используются при создании логфайлов?
9. Опишите процесс создания логфайлов на примере программы вычисления числа π.
10. Каково назначение функций MPE_Log_event(event1a, 0, "start broadcast"); MPE_Log_event(event1b, 0, "end broadcast")?
11. Каково назначение функций MPE_Log_event(event2a, 0, "start compute"); MPE_Log_event(event2b, 0, "end compute")?
12. Какие способы анализа логфайлов Вы знаете?
13. Какое различие между форматами логфайлов ALOG, CLOG и SLOG?
14. Какие средства просмотра логфайлов графического типа Вы знаете?
15. Объясните структуру изображения, создаваемого инструментом upshot?

Глава 7. ЭФФЕКТИВНОСТЬ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

7.1. СЕТЕВОЙ ЗАКОН АМДАЛА

Закон Амдала. Одной из главных характеристик параллельных систем является ускорение R параллельной системы, которое определяется выражением

$$R = T_1 / T_n,$$

где T_1 – время решения задачи на однопроцессорной системе; а T_n – время решения той же задачи на n-процессорной системе.

Пусть

$$W = W_{ск} + W_{пр},$$

где W – общее число операций в задаче; $W_{пр}$ – число операций, которые можно выполнять параллельно; $W_{ск}$ – число скалярных (нераспараллеливаемых) операций.

Обозначим также через t время выполнения одной операции. Тогда получаем известный закон Амдала [4, 5]:

$$R = \frac{Wt}{(W_{ск} + \frac{W_{пр}}{n})t} = \frac{1}{a + \frac{1-a}{n}} \xrightarrow[n\to\infty]{} \frac{1}{a}. \qquad (7.1)$$

Здесь $a = W_{ск}/W$ – удельный вес скалярных операций.

Закон Амдала определяет принципиально важные для параллельных вычислений положения.

1. Ускорение зависит от потенциального параллелизма задачи (величина $1 - a$) и параметров аппаратуры (числа процессоров n).
2. Предельное ускорение определяется свойствами задачи.

Пусть, например, $a = 0{,}2$ (что является реальным значением), тогда ускорение ≤ 5 при любом числе процессоров, т. е. максимальное ускорение определяется потенциальным параллелизмом задачи. Очевидной является чрезвычайно высокая чувствительность ускорения к изменению величины a.

Сетевой закон Амдала. Закон Амдала не отражает потерь времени на межпроцессорный обмен сообщениями. Эти потери могут не только снизить ускорение вычислений, но и замедлить вычисления по сравнению с однопроцессорным вариантом. Поэтому необходима некоторая модернизация выражения 7.1.

Перепишем формулу 7.1 следующим образом:

$$R_c = \frac{Wt}{(W_{ск} + \frac{W_{пр}}{n})t + W_c t_c} = \frac{1}{a + \frac{1-a}{n} + \frac{W_c t_c}{Wt}} = \frac{1}{a + \frac{1-a}{n} + c},$$

где W_c – количество передач данных; t_c – время одной передачи.

Выражение

$$R_c = \frac{1}{a + \frac{1-a}{n} + c}$$

является сетевым законом Амдала. Этот закон определяет следующие особенности многопроцессорных вычислений.

Коэффициент сетевой деградации вычислений:

$$c = \frac{W_c t_c}{Wt} = c_a c_t, \qquad (7.2)$$

определяет объем вычислений, приходящийся на одну передачу данных (по затратам времени). При этом c_a определяет алгоритмическую составляющую коэффициента деградации, обусловленную свойствами алгоритма, а c_t – техническую составляющую, которая зависит от соотношения технического быстродействия процессора и аппаратуры коммуникационной сети.

Следовательно для повышения скорости вычислений следует воздействовать на обе составляющие коэффициента деградации. Для многих задач и сетей коэффициенты c_a и c_t могут быть вычислены аналитически, хотя они определяются множеством факторов: алгоритмом задачи [14, 15], размером данных, реализацией функций обмена библиотеки MPI, использованием разделяемой памяти, техническими характеристиками коммуникационных сред и их протоколов.

Даже если задача обладает идеальным параллелизмом, сетевое ускорение определяется величиной

$$R_c = \frac{1}{\frac{1}{n} + c} = \frac{n}{1 + cn} \xrightarrow[c \to 0]{} n$$

и уменьшается при увеличении числа процессоров. Следовательно, сетевой закон Амдала должен быть основой оптимальной разработки алгоритма и программирования задач, предназначенных для решения на многопроцессорных ЭВМ.

В некоторых случаях используется еще один параметр для измерения эффективности вычислений – коэффициент утилизации:

$$z = \frac{R_c}{n} = \frac{1}{1 + cn} \xrightarrow[c \to 0]{} 1 \ .$$

7.2. ЭФФЕКТИВНОСТЬ И МАСШТАБИРУЕМОСТЬ ВЫЧИСЛЕНИЙ

В качестве примера оценки эффективности вычислений рассмотрим сетевую эффективность задачи умножения матрицы на вектор и умножение матрицы на матрицу.

Умножение матрицы на вектор. Алгоритм такого умножения состоит в следующем: квадратная матрица A размером $m \times m$ находится в

головном процессе, а вектор b находится во всех остальных процессах. После вычисления i-м процессом очередного элемента результата головной процесс передает ему новую строку матрицы A и так далее до исчерпания строк матрицы A.

Тогда для каждого элемента результирующего вектора C, получаемого от перемножения матрицы A на вектор b, нужно выполнить m умножений, и $m - 1$ сложений. Имеется m элементов вектора C, поэтому общее количество операций с плавающей точкой будет

$$m(m + (m - 1)) = 2m^2 - m.$$

Для простоты предполагаем, что на сложение и умножение затрачивается одинаковое количество времени, равное $T_{выч}$, и полное время вычисления в основном зависит от операций с плавающей точкой. Таким образом, грубая оценка времени на вычисления равна

$$(2m^2 - m)T_{выч}.$$

Оценим затраты на коммуникацию. Затраты на посылку b каждому подчиненному процессу не будем учитывать, предположим, что он прибывает туда другим способом (например, может быть там вычислен). Тогда количество чисел, которые должны быть переданы, равно $m + 1$ (m – чтобы послать строку матрицы A, а 1 – чтобы послать ответ назад) для каждого из m элементов C, что дает в результате

$$m(m + 1) = m^2 + m.$$

Если предположить, что время, требуемое на передачу числа с плавающей запятой, равняется $T_{ком}$, то полное время связи примерно равно $(m^2 + m)T_{ком}$. Поэтому отношение времени коммуникаций к времени вычислений согласно формуле 7.2 равно

$$C = C_a \times C_t = \left(\frac{m^2 + m}{2m^2 - m} \right) \times \left(\frac{T_{ком}}{T_{выч}} \right). \qquad (7.3)$$

Предположим, что:

1. для больших m (именно для таких задач нужны параллельные ЭВМ) можно пренебречь m по сравнению с m^2;
2. в качестве коммуникационной сети используется сеть Fast Ethernet с пропускной способностью V = 10 Мбайт/с;
3. объединенное время $T_{выч}$ (умножение, сложение, два обращения в память) требует 20 нс при частоте процессора 500 МГц;
4. длина слова с плавающей запятой может составлять 10 байтов.

Тогда, согласно выражению (7.3), получаем

$$C = C_a \times C_t = \frac{T_{ком}}{2T_{выч}} = \frac{10V}{2T_{выч}} = \frac{10/10^7}{2 \cdot 20 \cdot 10^{-9}} = 25.$$

Следовательно, даже при идеальном распараллеливании ($a = 0$):

$$R_c = \frac{1}{1/n + 25}$$

матрично-векторное умножение всегда будет выполняться в многопроцессорной системе с замедлением по отношению к однопроцессорному варианту, поэтому необходимы другие алгоритмы матрично-векторного умножения, обеспечивающие ускорение.

Умножение матрицы на матрицу. Для матрично-матричного умножения вектор b становится матрицей B, которая копируется в каждый процесс. Матрица A располагается в головном процессе и в процессе вычислений построчно рассылается в остальные процессы.

Процесс i получает строку матрицы от головного процесса, вычисляет строку результата и пересылает ее головному процессу. Вычисления заканчиваются после того, как будут обработаны все строки матрицы A.

Пусть A, B и C – квадратные матрицы. Тогда число операций для каждого элемента матрицы C будет (как и прежде) равно m умножений и $m - 1$ сложений, но теперь вычисляется m^2 элементов, а не m. Поэтому число операций с плавающей запятой

$$m(2m - 1) = 2m^3 - m^2.$$

Количество чисел с плавающей запятой в каждой строке равно m. На каждую строку необходимо сделать две пересылки (послать строку матрицы A и получить строку результата для матрицы C). В матрице A имеется m строк, следовательно, общее количество пересылаемых чисел при умножении матриц равно $m \times 2m$. Тогда отношение времени коммуникаций к времени вычислений с учетом формулы 7.2 равно

$$C = \left(\frac{2m^2}{2m^3 - m^2} \right) \times \left(\frac{T_{ком}}{T_{выч}} \right). \qquad (7.4)$$

Отношение стремится к $1/m$ при увеличении m. Поэтому для этой задачи мы должны ожидать, что коммуникационные перерасходы будут играть все меньшую роль при увеличении размера задачи.

С помощью рассуждений, которые проводились выше для векторно-матричного умножения, учитывая формулу 7.4 и необходимость посылки равного объема данных туда и обратно, получаем:

$$C = C_a \times C_t = \frac{1}{m}\frac{10V}{T_{выч}} = \frac{1}{m}\frac{10/V}{20 \cdot 10^{-9}}.$$

Поскольку для коммуникационной системы SCI (см. параграф 1.4) V = 80 Мбайт/с, то для сетей Fast Ethernet (FE) и SCI получаем:

$$R_c^{FE} = 1 \Big/ \left(\frac{1}{n} + \frac{100}{m}\right), \qquad (7.5) \qquad R_c^{SCI} = 1 \Big/ \left(\frac{1}{n} + \frac{12.5}{m}\right). \qquad (7.6)$$

Ниже для задачи умножения матрицы на матрицу приводятся два графика (рис. 7.1, 7.2), построенные по выражениям 7.5 и 7.6. Справа на рисунках указано число процессоров.

Реальные данные по производительности 36-процессорного кластера SCI и 40-процессорного кластера SKY, установленных в НИВЦ МГУ, приведены в [2].

Приведенные графики позволяют сделать следующие выводы.

1. Ускорение существенно зависит от объема вычислений, вызванного однократной передачей данных. Если это соотношение неудовлетворительное, то при любом числе процессоров происходит замедление, а не ускорение вычислений.
2. Сравнение графиков для Fast Ethernet и SCI также показывает сильную зависимость ускорения от пропускной способности коммуникационной сети, причем для SCI высокое ускорение достигается уже при меньших размерах матриц.

К эффективности вычислений также относится понятие «масштабирование». В общем случае под *масштабированием* понимают сохранение эффективности некоторой системы при увеличении ее размера. Для кластеров «масштабируемость является мерой, которая показывает, можно ли данную проблему решить быстрее, увеличив количество процессорных элементов» [13].

Введем понятие «диапазон масштабирования», он будет определять наибольшее количество процессоров, при котором коэффициент утилизации не уменьшается ниже заданного предела.

На рис.7.3 представлен график ускорения и коэффициента утилизации для гипотетического случая.

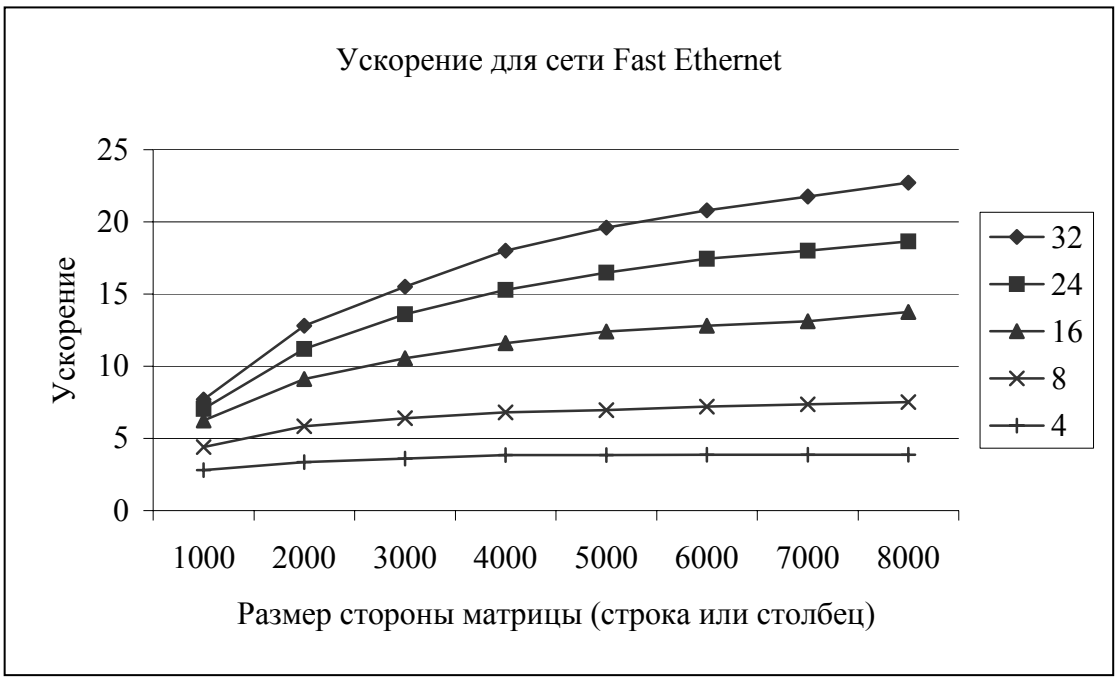


Рис. 7.1. Зависимость ускорения от размера матриц
и числа процессоров для сети Fast Ethernet

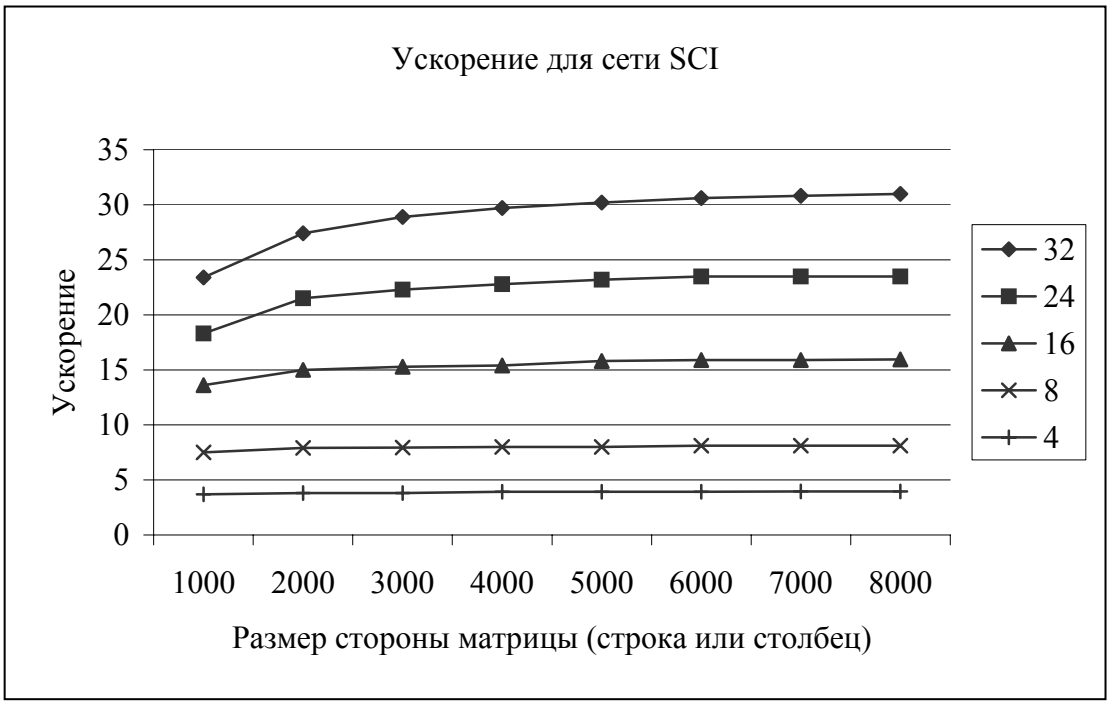


Рис. 7.2. Зависимость ускорения от размера матриц
и числа процессоров для сети SCI

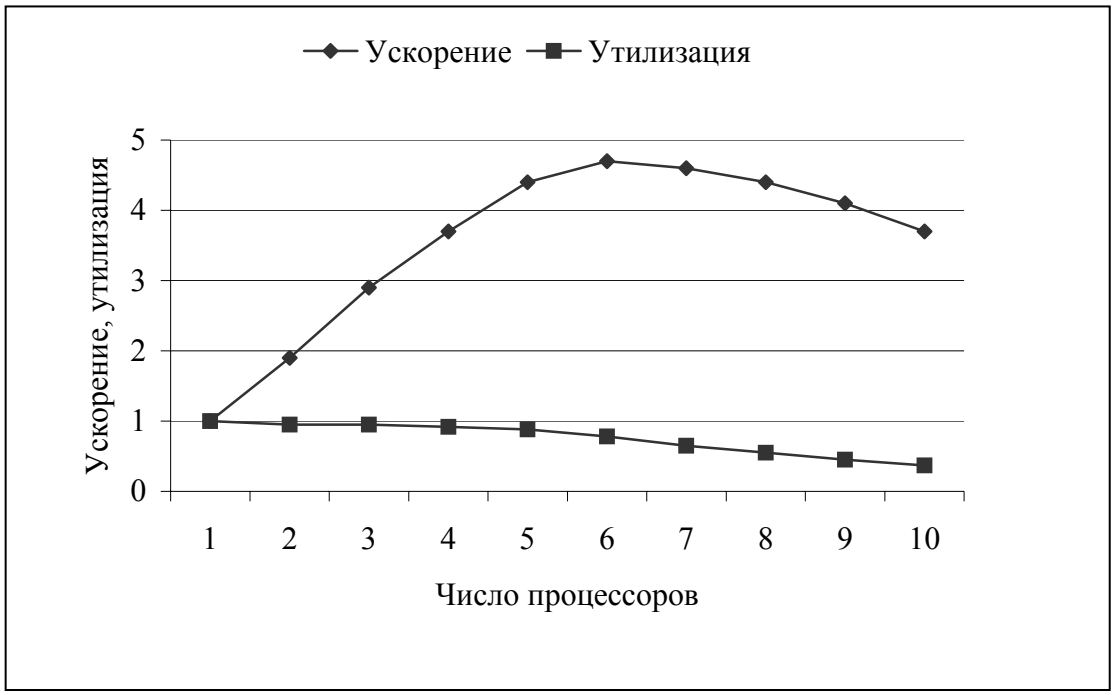


Рис.7.3. Зависимость коэффициента утилизации от числа процессоров

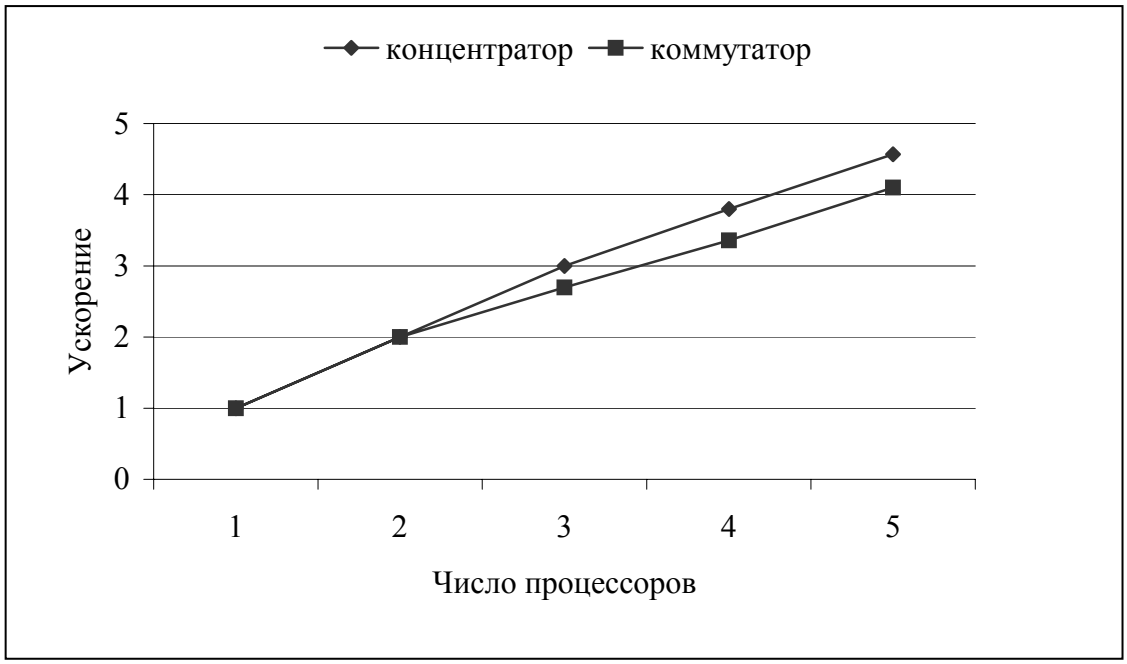


Рис. 7.4. Ускорение при использовании в кластере коммутатора и концентратора при решении задачи СЛАУ (метод Якоби)

Очевидно, что масштабирование сохраняется, если при увеличении числа процессоров ускорение растет (хоть и с замедлением) и при этом коэффициент утилизации не падает ниже заданной величины. Обычно считается, что масштабируемость еще сохраняется, если коэффициент утилизации не ниже 0,5.

Масштабирование необходимо исследовать в том случае, если программа отлаживается на кластере рабочих станций небольшого размера, а последующая эксплуатация выполняется на кластере типа СКИФ, где число процессоров значительно больше.

Эффективность вычислений зависит от того, какая аппаратура используется для коммуникаций в локальной сети. В концентраторе Ethernet используется единственная разделяемая среда, поэтому все операции обмена в кластере выполняются последовательно. В коммутаторе Ethernet эти операции выполняются параллельно, поэтому ускорение увеличивается, что показано на графике (рис.7.4).

7.3. СИСТЕМНЫЕ ПРОБЛЕМЫ

На характеристики программы влияют различные факторы, включая поведение кэша, наличие других пользователей на машине и др. Ниже представлены некоторые общие проблемы и способы их преодоления.

Задача слишком велика для физического размера памяти. При временной оценке программы всегда следует оставлять по крайней мере десять процентов запаса между общим объемом используемой процессом памяти и физическим размером памяти. Если программирование производится с использованием библиотеки PETSc, можно применить опцию -log_summary, которая выводит сумму памяти, используемой основными PETSc объектами, обеспечивая этим нижнюю границу использованной памяти.

Эффект других пользователей. Если другие пользователи выполняют работы на том же процессоре, на котором программа профилируется, оценки времени будут несостоятельными.

Накладные расходы оценки времени. На определенных машинах даже процедуры вызова системных часов медленные. Это искажает все оценки времени, поэтому необходимы тесты, определяющие время вызова системных часов.

Задача слишком велика для хороших характеристик кэша. Определенные машины с медленным доступом в память пытаются скомпенсировать это путем использования кэша. Поэтому если суще-

ственная часть приложения находится внутри кэша, программа будет достигать очень хороших характеристик. Если же программа слишком велика, характеристики могут деградировать. Чтобы оценить, влияет ли эта ситуация на определенную программу, можно пытаться представить быстродействие как функцию размера задачи. Если быстродействие в некоторой точке резко падает, возможно, задача слишком велика для кэша.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое ускорение параллельных вычислений?
2. Сформулируйте закон Амдала для многопроцессорной системы.
3. Определите ускорение для многопроцессорной системы, если удельный вес скалярных операций равен 0,1.
4. Что определяет закон Амдала?
5. Какую характеристику определяет сетевой закон Амдала?
6. Что такое коэффициент сетевой деградации?
7. Какие факторы влияют на эффективность сетевых вычислений?
8. Определите коэффициент утилизации, если коэффициент сетевой деградации равен 0,5, число процессоров равно 10.
9. Определите коэффициент сетевой деградации для задачи умножения вектора на матрицу, если размерность матрицы равна 1000 ×1000, время, требуемое на передачу числа с плавающей запятой равняется 100 нс, время на операцию с плавающей запятой равно 10 нс.
10. Определите коэффициент сетевой деградации для задачи умножения матрицы на матрицу, если размерность матрицы равна 1000 ×1000, время, требуемое на передачу числа с плавающей запятой равняется 100 нс, время на операцию с плавающей запятой равно 10 нс.
11. Какие выводы можно сделать из изучения графиков умножения матриц для сети Fast Ethernet и сети SCI?
12. Что такое масштабирование вычислений?
13. Определите понятие диапазон масштабирования вычислений.
14. Как связаны между собой коэффициент утилизации и диапазон масштабирования?
15. Как влияет на ускорение использование коммутатора вместо концентратора?
16. Перечислите основные системные проблемы, влияющие на время выполнения программы.

III. **БИБЛИОТЕКА PETSC**

Глава 8. НАЧАЛЬНЫЕ СВЕДЕНИЯ

8.1. СОСТАВ БИБЛИОТЕКИ

Библиотека **PETSc** (**P**ortable **E**xtensible **T**oolkit for **S**cientific **C**omputation) предназначена для разработки крупномасштабных приложений на языках Fortran, C и C++. PETSc включает различные компоненты, часть из которых рассматривается в этом разделе. Каждый компонент имеет дело с частным семейством объектов (например, с векторами) и операциями, которые нужно выполнять над этими объектами. Объекты и операции в PETSc определены на основе долгого опыта научных вычислений.

Рис. 8.1 представляет состав компонентов библиотеки и их иерархическую организацию, рис. 8.2 – состав используемых численных методов. Используемые численные методы описаны в [14–19].

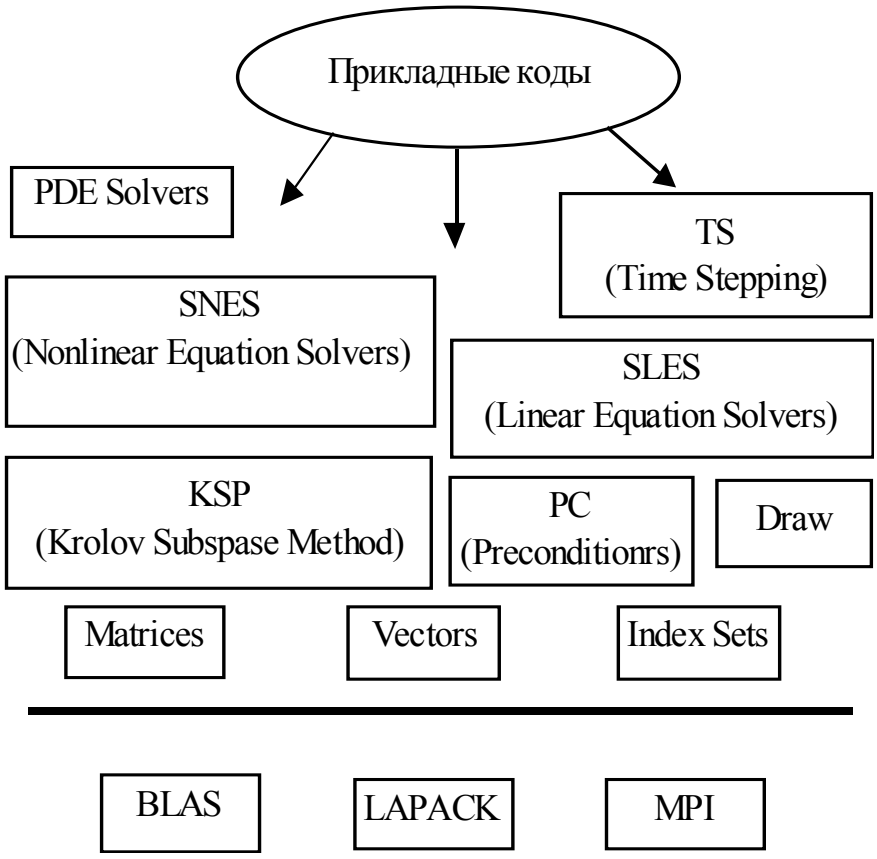


Рис.8.1. Состав компонентов библиотеки PETSc

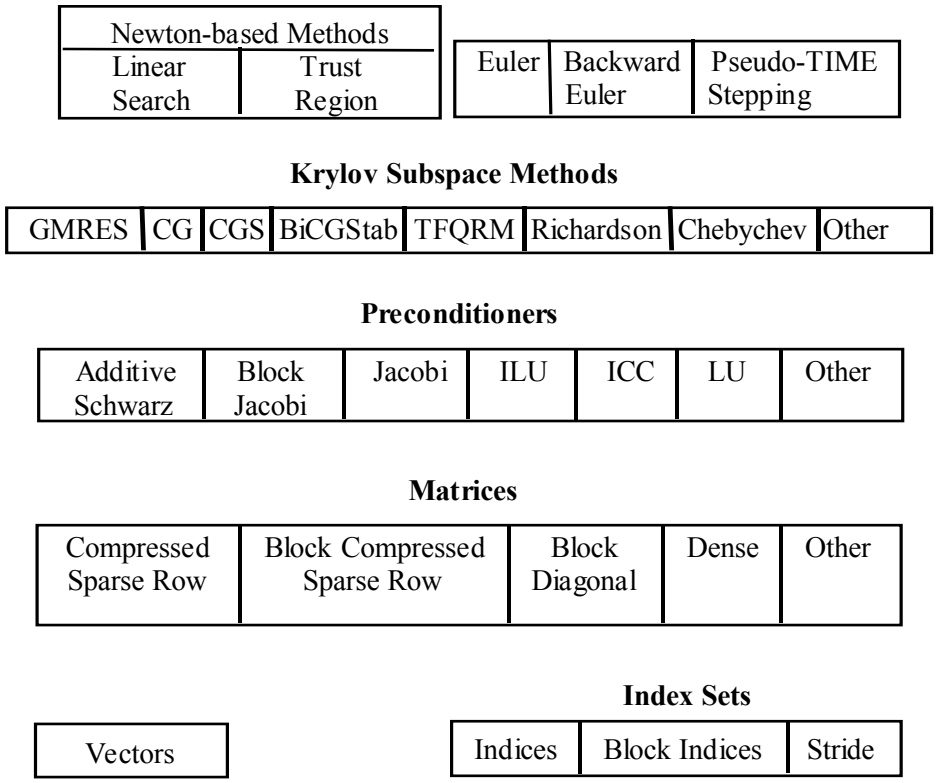


Рис. 8.2. Численные методы, реализованные в PETSc

Библиотека PETSc содержит следующие методы и объекты:

- индексные ряды для индексации внутри вектора, переименования и т. д.;
- векторы;
- матрицы (в общем случае разреженные);
- распределенные массивы (полезны для параллелизации задач на основе сеток);
- методы подпространств Крылова;
- предварительную обработку;
- нелинейные методы;
- пошаговую временную обработку для решения времязависимых дифференциальных уравнений в частных производных (ДУЧП).

Последние версии программного пакета PETSc можно получить с сайта [1]. Пакет занимает более 50 Мбайт памяти.

Исходный текст описания PETSc на английском языке находится на сайте *http://www-unix.mcs.anl.gov/petsc*. Полный перевод этого документа представлен на сайте [3]. В настоящем пособии представлена информация только по векторам, матрицам и методам решения систем

линейных алгебраических уравнений (СЛАУ). Решения СЛАУ реализуются с помощью компонента SLES (**L**inear **E**quation **S**olvers – решатели систем линейных алгебраических уравнений).

8.2. ЗАПУСК PETSC-ПРОГРАММ

Все PETSc-программы базируются на стандарте MPI. Поэтому, чтобы выполнить PETSc-программу, пользователь обязан знать процедуру запуска MPI-программы на избранной компьютерной системе. Например, при использовании реализации MPICH для MPI программу инициирует следующая команда, которая запускает восемь процессов:

mpirun -np 8 petsc_program_name petsc_options

Все PETSc-программы поддерживают ряд опций. Полный список этих опций можно быть получен при запуске любой программы PETSc с опцией -help.

8.3. НАПИСАНИЕ PETSC-ПРОГРАММ

Большинство программ PETSc начинаются обращением

PetscInitialize(int *argc,char ***argv,char *file,char *help);

которое инициализирует PETSc и MPI. Аргументы **argc** и **argv** являются аргументами командной строки и используются во всех C и C++ программах. Файл аргументов в качестве опции указывает на альтернативное имя файла PETSc-опций, который размещается в пользовательской директории. Аргумент help есть символьная строка, которая будет выводиться, если программа выполняется с опцией –help.

Процедура PetscInitialize() автоматически вызывает процедуру MPI_Init(), если она не была прежде инициализирована. В определенных обстоятельствах, когда нужно инициализировать MPI непосредственно (или инициализация производится некоторой другой библиотекой), пользователь сначала вызывает MPI_Init() (или это делает другая библиотека), а затем выполняется обращение к PetscInitialize(). По умолчанию процедура PetscInitialize() добавляет к коммуникатору MPI_COMM_WORLD коммуникатор с фиксированным именем PETSC_COMM_WORLD. В большинстве случаев пользователю достаточно коммуникатора PETSC_COMM_WORLD, чтобы указать все процессы данного приложения, а PETSC_COMM_SELF указывает на одиночный процесс. Пользователи, которым нужны процедуры PETSc для подмножества процессов внутри большой работы, или, если необходимо использовать «master» процесс для координации работы

«slave» процессов, должны описать альтернативный коммуникатор для PETSC_COMM_WORLD обращением

PetscSetCommWorld(MPI_Comm comm);

Это необходимо сделать до вызова процедуры PetscInitialize(), но после вызова процедуры MPI_Init(). Процедура PetscSetCommWorld() в большинстве случаев вызывается однажды в процессе.

Все процедуры PETSc возвращают целое значение, указывающее, имела ли место ошибка при вызове. Если ошибка была обнаружена, код ошибки принимает ненулевое значение, в противном случае устанавливается нуль. Для языков C/C++ ошибка есть значение, возвращаемое процедурой.

Все PETSc-программы должны вызывать процедуру PetscFinalize() как их конечное предложение:

PetscFinalize();

Эта процедура вызывает MPI_Finalize(), если MPI был запущен вызовом PetscInitialize(). Если MPI был запущен внешним по отношению к PETSc образом, например, пользователем, то пользователь несет ответственность за вызов MPI_Finalize().

Include-файлы. C/C++ включает файлы для PETSc, которые следует использовать с помощью предложения

#include "petscsles.h"

где petscsles.h есть include-файл для компонента SLES.

Каждая программа PETSc обязана описать файл, который соответствует наиболее высокому уровню объектов PETSc, необходимых внутри программы. Все необходимые файлы нижнего уровня включаются автоматически внутрь файлов верхнего уровня. Например, petscsles.h включает petscmat.h (matrices), petscvec.h (vectors) и petsc.h (основной PETSc-файл). Все include-файлы PETSc размещены в директории *${PETSC_DIR}/include*.

Векторы. Вектор x размером M создается командой

VecCreate(MPI_Comm comm,int m,int M,Vec *x);

где comm обозначает коммуникатор MPI. Тип памяти для этого вектора может быть установлен обращением к процедуре VecSetType() или VecSetFromOptions(). Дополнительные векторы того же типа могут быть сформированы с помощью вызова

VecDuplicate(Vec old,Vec *new);

Команды

```
VecSet(PetscScalar *value,Vec x);
VecSetValues(Vec x,int n,int *indices,PetscScalar *values,INSERT_VALUES);
```

устанавливают все компоненты вектора равным соответственно некоторому скалярному значению или различным скалярным значениям.

Обратим внимание на использование переменной типа PetscScalar. Она определяется как тип double в C/C++ для версий PETSc, которые не были откомпилированы для использования с комплексными числами. Тип PetscScalar определяется соответственно, когда библиотека PETSc компилируется для использования с комплексными числами.

Матрицы. Использование матриц и векторов в PETSc сходно. Пользователь может создавать новую матрицу A, которая имеет M строк и N столбцов с помощью процедуры

```
MatCreate(MPI_Comm comm, int m, int n, int M, int N, Mat *A);
```

где формат матрицы может быть описан на этапе выполнения. Пользователь может альтернативно описывать для каждого номера процесса локальную строку и столбец, используя m и n. Значения затем могут быть установлены командой

```
MatSetValues(Mat A, int m,int *im, int n, int *in,
                    Scalar *values, INSERT_VALUES);
```

После того как все элементы помещены в матрицу, она должна быть обработана парой команд:

```
MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

Программы для решения систем линейных уравнений. После создания матрицы и векторов, которые определяют линейную систему $Ax = b$, пользователь затем может решить эту систему с помощью SLES, применив следующую последовательность команд:

```
SLESCreate(MPI_Comm comm,SLES *sles);
SLESSetOperators(SLES sles,Mat A,Mat PrecA,MatStructure flag);
SLESSetFromOptions(SLES sles);
SLESSolve(SLES sles,Vec b,Vec x,int *its);
SLESDestroy(SLES sles);
```

Пользователь сначала создает контекст SLES и определяет операторы, связанные с системой. Затем он устанавливает различные опции для настройки решения, решает линейную систему и, наконец, удаляет контекст SLES. Обратим внимание на команду, которая разрешает

пользователю настраивать метод линейного решения на этапе выполнения, используя набор опций SLESSetFromOptions(). Используя этот набор, пользователь может не только избрать итерационный метод и предобработку, но и определить устойчивость сходимости, установить различные мониторинговые процедуры.

Проверка ошибок. Все процедуры PETSc возвращают целое значение, отмечая наличие или отсутствие ошибки во время выполнения вызова. Макрос CHKERRQ(ierr) проверяет значение ierr и при обнаружении ошибки вызывает обработчик ошибок. CHKERRQ(ierr) следует использовать во всех процедурах, чтобы установить полный список ошибок. На рис. 8.3 представлен трек, сгенерированный детектором ошибки внутри PETSc-программы.

```
eagle>mpirun -np 1 ex3 -m 10000
[0]PETSC ERROR: MatCreateSeqAIJ() line 1673 in src/mat/impls/aij/seq/aij.c
[0]PETSC ERROR: Out of memory. This could be due to allocating
[0]PETSC ERROR: too large an object or bleeding by not properly
[0]PETSC ERROR: destroying unneeded objects.
[0]PETSC ERROR: Try running with -trdump for more information.
[0]PETSC ERROR: MatCreate() line 99 in src/mat/utils/gcreate.c
[0]PETSC ERROR: main() line 71 in src/sles/examples/tutorials/ex3.c
[0] MPI Abort by user Aborting program !
[0] Aborting program!
p0_28969: p4_error: : 1
```

Рис. 8.3. Пример трассировки ошибок

Ошибка обнаружена в файле *${PETSC_DIR}/src/mat/impls/aij/seq/aij.c,* и вызвана попыткой разместить слишком большой массив в памяти.

При запуске отладочной версии библиотеки PETSc важно проверить, не испорчена ли память (внешней записью или нарушением границ и т. д.). Макрос CHKMEMQ для этой проверки можно вызвать в любом месте программы. Размещая несколько таких макросов в программе, можно легко отследить, в каком месте программы была повреждена информация в памяти.

Профилирование. Профилирование в общем случае означает определение каких-либо характеристик системы. По отношению к параллельным вычислениям профилирование, как правило, используется для изучения эффективности этих вычислений (времени вычислений, ускорения, коэффициента утилизации и др.). Пользователю не стоит тратить время на оптимизацию алгоритма или программы, пока он не установит, на что уходит основное время при расчетах в задаче реального размера.

Если прикладная программа скомпилирована с помощью флага (который по умолчанию есть во всех версиях) DPETSC_USE_LOG, тогда между вызовами процедур PetscInitialize() и PetscFinalize() могут быть активированы различные виды профилирования во время исполнения программы. Заметим, что флаг DPETSC_USE_LOG может быть описан при инсталляции PETSc в файле *${PETSC_DIR}/bmake/${PETSC_ARCH}/base_variables*.

Имеются следующие опции профилирования:

- -log_summary – печатает ASCII-версию характеристик по окончании программы. Данные носят компактный характер и требуют небольших накладных расходов, поэтому -log_summary является первичным средством мониторинга характеристик PETSc-программ;
- -log_info – выводит на экран обширную информацию о программе. Эта опция предоставляет детали об алгоритме, структуре данных и т. д. Поскольку получение такой информации замедляет вычисления, эту опцию не следует использовать во время оценки характеристик программ;
- -log_trace[logfile] – трассы начала и окончания всех PETSc-событий. Эта опция, используемая в соединении с -log_info, полезна, когда нужно увидеть без запуска отладчика, где зависла программа.

Некоторые методы отладки. При разработке большой программы часто бывает, что программа работает корректно, но последующие изменения кода дают другой результат по самой неизвестной причине. При этом даже точное определение точки, в которой старая и новая программы расходятся, является проблемой.

В других случаях программа генерирует различные результаты, когда выполняется на другом количестве процессов, хотя должен быть один и тот же результат.

PETSc обеспечивает некоторую поддержку для определения места в программе, где вычисления привели к различию в результатах. Прежде всего, нужно скомпилировать обе программы с разными именами. Затем обе программы запускаются на выполнение как одна MPI-программа. Эта операция зависит от MPI-реализации. Например, когда используется MPICH на рабочих станциях, могут быть использованы файлы procgroup, чтобы указать процессы, в которых должна быть выполнена работа.

Поэтому чтобы запустить две программы old и new, каждую на двух процессах, следует создать файл procgroup со следующим содержанием:

local 0
workstation1 1 /home/bsmith/old
workstation2 1 /home/bsmith/new
workstation3 1 /home/bsmith/new

Затем можно выполнить команду

mpirun -p4pg <procgroup_filemame> old -compare <tolerance> [options]

Заметим, что одинаковые опции времени исполнения должны быть использованы для двух программ. В первый раз, когда внутренний результат или норма определят расхождение, большее, чем <tolerance>, PETSc будет генерировать ошибку.

Могут быть использованы обычные опции времени исполнения -start_in_debugger и -on_error_attach_debugger. Пользователь может также поместить команды

PetscCompareDouble()
PetscCompareScalar()
PetscCompareInt()

в частях прикладного кода, чтобы проверить соответствие между двумя версиями.

8.4. ПРИМЕР ПРОСТОЙ ПАРАЛЛЕЛЬНОЙ PETSC-ПРОГРАММЫ

Поскольку PETSc использует MPI для всех межпроцессных обменов, пользователь свободен в использовании процедур MPI в своем приложении. Однако, по умолчанию, пользователь изолирован от многих деталей передачи сообщений внутри PETSc, поскольку они невидимы внутри параллельных объектов, таких как векторы, матрицы и решатели. В дополнение PETSc обеспечивает средства, такие, как обобщенные векторные рассылки/сборки и распределенные массивы, чтобы помочь в управлении параллельными данными.

Пользователь обязан описывать коммуникатор при создании любого объекта PETSc (векторы, матрицы, решатели), чтобы указать процессы, на которые распределяется объект. Например, некоторые команды для создания матриц, векторов и линейных решателей:

MatCreate(MPI_Comm comm,int M,int N,Mat *A);
VecCreate(MPI_Comm comm,int m,int M,Vec *x);
SLESCreate(MPI_Comm comm,SLES *sles);

Процедуры построения объектов являются коллективными. Это означает, что все процессы в коммуникаторе обязаны обращаться к этим процедурам. Кроме того, если используется последовательность

коллективных процедур, они обязаны вызываться в том же самом порядке на каждом процессе.

Следующий пример (рис. 8.4) иллюстрирует параллельное решение линейной системы уравнений. Этот код находится в файле *${PETSC_DIR}/src/sles/examples/tutorials/ex2.c*, обрабатывает двухмерный лапласиан, дискретизованный конечными разностями, линейная система решается с помощью SLES.

```
/*$Id: ex2.c,v 1.92 2001/03/23 23:23:55 balay Exp $*/
/* Запуск программы: mpirun -np <procs> ex2 [-help] [all PETSc options]
*/
static char help[] = "Solves a linear system in parallel with SLES.\n\
Input parameters include:\n\
       -random_exact_sol : solution vector\n\
       -view_exact_sol   : write exact solution vector to use a random exact stdout\n\
       -m <mesh_x>       : number of mesh points in x-direction\n\
       -n <mesh_n>       : number of mesh points in y-direction\n\n";
/* В качестве примера далее приводится решение двумерного Лапласиа
    с  помощью функций SLES
*/
/* подключаем  "petscsles.h" , чтобы можно было использовать решатели SLES.
Заметим, что это автоматически подключает
petsc.h          - base PETSc routines    petscvec.h   - vectors
petscsys.h       - system routines        petscmat.h   - matrices
petscis.h        - index sets             petscksp.h   - Krylov subspace methods
petscviewer.h    - viewers                petscpc.h    - preconditioners
 */

#include "petscsles.h"
#undef __FUNCT__
#define __FUNCT__ "main"

int main(int argc,char **args)
{ Vec x,b,u;                    /* приближенное решение, RHS, точное решение */
  Mat A;                        /* матрица линейной системы */
  SLES sles;                    /* метод решения линейной системы */
  PetscRandom rctx;             /*  генератор случайных чисел */
  double norm;                  /* норма ошибки решения */
  int i,j,I,J,Istart,Iend,ierr,m = 8,n = 7,its;
  PetscTruth flg;
  Scalar v,one = 1.0,neg_one = -1.0;
  KSP ksp;

 PetscInitialize(&argc,&args,(char *)0,help);
 ierr = PetscOptionsGetInt(PETSC_NULL,"-",&m,PETSC_NULL);CHKERRQ(ierr);
 ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
```

```
/*  Организуем матрицу и вектор свободных членов, который задает
    линейную  систему Ax = b
*/
/*
  Создаем параллельную матрицу, описывая только ее глобальные размеры. Когда используется MatCreate(), формат матрицы может быть описан на этапе исполнения. Разбиение параллельно матрицы на части по процессорам библиотекой PETSc также относится на время исполнения. Замечание по эффективности: для задач большого размера предварительное распределение матричной памяти является решающим для получения хорошей эффективности. Поскольку процедура MatCreate() не допускает предварительного распределения памяти, рекомендуется для практических задач вместо нее использовать процедуры для создания частных матричных форматов, например, MatCreateMPIAIJ() - parallel AIJ (compressed sparse row) MatCreateMPIBAIJ() - parallel block AIJ. Детали – в разделе о матрицах.
*/
ierr=MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,
                    m*n,m*n,&A); CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);
/*
  Матрица разделена крупными кусками строк по процессам. Определяем, какие строки матрицы размещены локально.
*/

ierr = MatGetOwnershipRange(A,&Istart,&Iend); CHKERRQ(ierr);
/*
  Размещаем матричные элементы. Каждому процессу нужно разместить только те элементы, которые принадлежат ему локально (все нелокальные элементы будут посланы в соответствующий процесс во время сборки  матрицы).
*/

for (I=Istart; I<Iend; I++)
{  v = -1.0;    i = I/n;    j = I - i*n;
    if (i>0)
    { J = I - n;
      ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);
    }
    if (i<m-1)
    { J = I + n;
      ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);
    }
    if (j>0)
    { J = I - 1;
      ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);
    }
    if (j<n-1)
    { J = I + 1;
```

```
      ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);
    }
    v = 4.0;
    ierr = MatSetValues(A,1,&I,1,&I,&v,INSERT_VALUES);CHKERRQ(ierr);
  }
  /*
    Сборка матрицы производится в два этапа: MatAssemblyBegin(), затем
    MatAssemblyEnd(). Если разместить отрезок программы между этими этапа-
    ми, то вычисления будут совмещаться с обменом.
  */
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  /*
    Создаем параллельные векторы. Формируем вектор и затем дублируем, если
    необходимо. Для VecCreate() и VecSetFromOptions(), указывается только гло-
    бальный размер вектора; параллельное разбиение определяется на этапе ис-
    полнения. Когда решается линейная система, векторы и матрицы обязаны быть
    разбиты соответственно.PETSc автоматически генерирует соответствующее
    разбиение матриц и векторов, когда MatCreate() и VecCreate() используются с
    тем же самым коммуникатором. Пользователь может альтернативно описать
    размеры локальных векторов и матриц, когда необходимо более сложное раз-
    биение путем замещения аргумента PETSC_DECIDE в VecCreate()).
  */
  ierr = VecCreate(PETSC_COMM_WORLD,PETSC_DECIDE,m*n,&u);
  CHKERRQ(ierr);
  ierr = VecSetFromOptions(u);CHKERRQ(ierr);
  ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
  ierr = VecDuplicate(b,&x);CHKERRQ(ierr);
  /*
    Установим точное решение, затем вычислим правосторонний вектор. По умол-
    чанию используем точное решение вектора для случая, когда все элементы
    вектора равны единице. Альтернативно, используя опцию -random_sol, фор-
    мируем решение вектора со случайными компонентами.
*/
  ierr = PetscOptionsHasName(PETSC_NULL,"-random_exact_sol",&flg);
         CHKERRQ(ierr);
  if (flg)
  {
   ierr =PetscRandomCreate(PETSC_COMM_WORLD,RANDOM_DEFAULT,&rctx);
         CHKERRQ(ierr);
   ierr = VecSetRandom(rctx,u);CHKERRQ(ierr);
   ierr = PetscRandomDestroy(rctx);CHKERRQ(ierr);
  }
```

```
else   ierr = VecSet(&one,u);CHKERRQ(ierr);
ierr = MatMult(A,u,b);CHKERRQ(ierr);
/*
  Выводим точное решение вектора, если необходимо
*/
ierr = PetscOptionsHasName(PETSC_NULL,"-view_exact_sol",&flg);
       CHKERRQ(ierr);
if (flg)
{  ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);  }
/*
  Создаем метод решения системы линейных уравнений
*/
ierr = SLESCreate(PETSC_COMM_WORLD,&sles);CHKERRQ(ierr);
/*
  Устанавливаем операторы. Здесь матрица, которая определяет линейную сис-
  тему,  служит как переобусловливающая матрица.
 */
ierr = SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN);
       CHKERRQ(ierr);
/*
  Устанавливается  метод решения. Определив из контекста SLES контекст KSP
  и PC, мы можем затем прямо вызвать любую KSP и PC процедуру, чтобы уста-
  новить различные опции.  Две следующие процедуры можно применять по вы-
  бору.
*/
ierr = SLESGetKSP(sles,&ksp);CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-2/((m+1)*(n+1)),1.e-50,PETSC_DEFAULT,
                                PETSC_DEFAULT);CHKERRQ(ierr);
/*
  Устанавливаем опции времени исполнения, например -ksp_type <type>,
   -pc_type <type>,   -ksp_monitor,   -ksp_rtol <rtol>. Эти опции будут заменять оп-
  ции, описанные выше, до тех пор пока после  любой настроечной процедуры
  вызывается SLESSetFromOptions()
*/
ierr = SLESSetFromOptions(sles);CHKERRQ(ierr);
/*
  Решается линейная система
  */
ierr = SLESSolve(sles,b,x,&its);CHKERRQ(ierr);
/*
  Проверяется решение  norm *= sqrt(1.0/((m+1)*(n+1)));
*/
ierr = VecAXPY(&neg_one,u,x);CHKERRQ(ierr);
ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
```

```
  /*
    Печатается информация о сходимости.  PetscPrintf() создает одно предложение
    печати из всех процессов, которые разделяют коммуникатор. Альтернативой
    является PetscFPrintf(), которая печатает в файл.
  */
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A iterations %d\n",
                     norm,its);  CHKERRQ(ierr);
  /*
    Освобождается рабочее пространство. Все PETSc объекты должны быть раз-
    рушены, когда они больше не нужны.
  */

  ierr = SLESDestroy(sles);CHKERRQ(ierr);
  ierr = VecDestroy(u);CHKERRQ(ierr); ierr = VecDestroy(x);CHKERRQ(ierr);
  ierr = VecDestroy(b);CHKERRQ(ierr); ierr = MatDestroy(A);CHKERRQ(ierr);
/*
    Всегда следует перед выходом из программы вызывать PetscFinalize(). Эта
    процедура заканчивает использование библиотеки PETSc и, как и в MPI, пре-
    доставляет обобщенную и диагностическую информацию, если установлены
    определенные опции времени исполнения (например, -log_summary).
*/
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

Рис. 8.4. Пример мультипроцессного PETSc кода

Директория *${PETSC_DIR}/src/<component>/examples/tutorials* содержит много примеров, которые могут служить образцами для создания пользовательских программ. Переменная <component> обозначает любые PETSc-компоненты, такие как snes или sles. Справочные страницы на *${PETSC_DIR}/docs/index.html* или *ttp://www.mcs.anl.gov/petsc/docs* обеспечивают индексацию к примерам в учебнике.

Чтобы написать новое приложение с использованием PETSc, можно предложить следующий порядок.

1. Инсталлировать и протестировать PETSc согласно инструкциям на PETSc web site.
2. Скопировать один из примеров в директорию компонентов, которая соответствует классу решаемой задачи (например, для линейных решателей *${PETSC_DIR}/src/sles/examples/ tutorials*).
3. Скопировать соответствующий makefile внутри директории примера, скомпилировать и запустить программу примера.
4. Использовать программу примера как стартовую точку для создания собственной программы.

8.5. СТРУКТУРА КОРНЕВОЙ ДИРЕКТОРИИ

Корневая директория PETSc имеет следующие поддиректории:

- **docs** – содержит все документы PETSc. В файле manual.pdf находядся гиперсвязи, удобные для печати или просмотра.
- **bin** – утилиты и короткие скрипты для использования с PETSc, включая petsarch (для установки переменной среды PETSC_ARCH);
- **bmake** – директория основного PETSc makefile. Включает поддиректории для различных архитектур;
- **include** – содержит все include файлы PETSc, которые видимы пользователю;
- **include/finclude** – содержит include файлы PETSc для языка Fortran;
- **include/pinclude** – частные PETSc include файлы, которые не следует использовать прикладным программистам;
- **src** – исходные коды для всех компонентов PETSc, которые включают: **vec** – векторы, **mat** – матрицы, **da** – распределенные массивы, **sles** – полный решатель линейных уравнений, **snes** – нелинейные решатели, **ts** – решатели для времязависимых систем дифференциальных уравнений, **sys** – общие системные процедуры, **fortran** – Fortran интерфейс.

Каждый исходный код PETSc имеет следующие поддиректории:

- **examples** – примеры программ для компонентов, включая: **tutorials** – программы для обучения пользователей PETSc, могут служить образцами для проектирования собственных приложений; **tests** – тестовые программы, не предназначены для изучения пользователями;
- **interface** – программы интерфейса к компонентам;
- **impls** – исходные коды для одной или нескольких реализаций;
- **utils** – утилиты. Поставщик здесь может знать о конкретной реализации, но идеальный вариант, когда он не знает о реализациях для других компонент.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для чего предназначена библиотека PETSc?
2. Каков состав библиотеки?
3. Какие математические методы используются в библиотеке PETSc?
4. Для каких языков предназначена библиотека?
5. Можно ли запустить PETSc-приложение как последовательную программу?
6. Как запускаются PETSc-программы?

7. Опишите структуру и обрамляющие операторы PETSc-программы.
8. Как создать новый коммуникатор в PETSc-приложении?
9. Какие структуры данных используются в PETSc-программе?
10. Как выглядит программа для решения СЛАУ?
11. Для чего используются SLES-операторы?
12. Как в PETSc производится трассировка ошибок?
13. Как в PETSc производится профилирование?
14. Постройте блок-схему параллельной программы, представленную на рис. 8.4.
15. Как размещаются элементы матрицы в параллельной программе решения СЛАУ на рис. 8.4 по процессам?
16. Что содержит корневая директория пакета PETSc?

Глава 9. ВЕКТОРЫ И ПАРАЛЛЕЛЬНОЕ РАСПРЕДЕЛЕНИЕ ДАННЫХ

9.1. СОЗДАНИЕ И СБОРКА ВЕКТОРОВ

Вектор (обозначим через Vec) – один из простейших объектов PETSc. Векторы используются для решения ДУЧП, для хранения свободных членов и результатов решения СЛАУ и др.

PETSc поддерживает два базисных векторных типа: последовательный и параллельный (для MPI). Чтобы создать последовательный вектор с m компонентами, можно использовать команду

```
VecCreateSeq(PETSC_COMM_SELF,int m,Vec *x);
```

Чтобы создать параллельный вектор, можно описать либо число компонентов, которые будут храниться на каждом процессе, либо передать это решение PETSc. Команда

```
VecCreateMPI(MPI_Comm comm,int m,int M,Vec *x);
```

создает вектор, который распределен по всем процессам в коммуникаторе comm, где m указывает число компонентов вектора, которые нужно хранить в локальном процессе, а M – общее число компонентов вектора. При автоматическом выборе с помощью PETSC_DECIDE нужно указать только локальную или глобальную размерность вектора, но не обе вместе. В общем случае можно использовать процедуры

```
VecCreate(MPI_Comm comm,int m,int M,Vec *v);
VecSetFromOptions(v);
```

которые автоматически генерируют соответствующий векторный тип (последовательный или параллельный) во всех процессах коммуникатора comm. Опция -vec_type mpi вместе с процедурами VecCreate() и

VecSetFromOptions() нужна для описания использования векторов даже для последовательного случая.

Все процессы в коммуникаторе comm обязаны вызывать процедуры создания векторов, поскольку эти процедуры коллективные для всех процессов коммуникатора. Если используется последовательность процедур VecCreateXXX(), они обязаны вызываться в том же порядке на каждом процессе коммуникатора.

Единственное значение всем компонентам вектора можно присвоить с помощью команды

VecSet(PetscScalar *value,Vec x);

Присвоение разных значений компонентам вектора – более сложный процесс, если нужно получить эффективный параллельный код. Присвоение ряду компонентов вектора значений есть двухшаговый процесс. Сначала вызываем процедуру

VecSetValues(Vec x,int n,int *indices,PetscScalar *values,INSERT_VALUES);

любое число раз в одном или всех процессах. Аргумент *n* дает количество компонент, установленных при этом присвоении. Целочисленный массив *indices* содержит индексы глобальных компонентов и его значением является массив индексов, которые должны быть заполнены. Любой процесс может установить любой компонент вектора, при этом PETSc гарантирует, что они автоматически сохранятся в правильной ячейке. Когда все значения уже размещены с помощью VecSetValues(), нужно вызвать

VecAssemblyBegin(Vec x);
VecAssemblyEnd(Vec x);

чтобы выполнить необходимую передачу сообщений для нелокальных компонент. Чтобы допустить возможность совмещения обменов и вычислений, код пользователя может выполнять любую последовательность действий между этими двумя вызовами, пока сообщения находятся в состоянии передачи.

Часто необходимо не заменить значения элементов вектора, а сложить соответствующие значения. Этот процесс осуществляется командой

VecSetValues(Vec x,int n,int *indices, PetscScalar *values,ADD_VALUES);

После добавки всех значений нужно обязательно вызывать процедуры сборки VecAssemblyBegin() и VecAssemblyEnd(). PETSc не по-

зволяет использовать INSERT_VALUES и ADD_VALUES одновременно, так как это приводит к недетерминированному поведению.

Вектор можно исследовать с помощью команды

VecView(Vec x,PetscViewer v);

Чтобы вывести вектор на экран, используется просмотрщик VIEWER_STDOUT_WORLD, который гарантирует, что параллельные векторы в stdout будут представляться правильно. Чтобы представить вектор в окне, используют установленный по умолчанию оконный просмотрщик VIEWER_DRAW_WORLD. Просмотрщик также можно создать с помощью процедуры VeiwerDrawOpenX(). Чтобы создать новый вектор того же формата, что и существующий, используется команда

VecDuplicate(Vec old,Vec *new);

Чтобы создать несколько новых векторов того же формата, что и существующий, используется команда

VecDuplicateVecs(Vec old,int n,Vec **new);

Эта процедура создает массив указателей на векторы. Две указанные процедуры полезны, поскольку они создают библиотечный код, который не зависит от формата используемых векторов. Кроме того, подпрограммы могут автоматически создавать рабочие векторы, основанные на описании существующего вектора. Когда вектор больше не нужен, он удаляется командой

VecDestroy(Vec x);

Чтобы удалить массив векторов, используется команда

VecDestroyVecs(Vec *vecs,int n);

Можно создать векторы, которые используют массив, созданный пользователем, а не с использованием внутреннего пространства массивов PETSc. Такие векторы могут быть созданы процедурами

VecCreateSeqWithArray(PETSC_COMM_SELF,int m,Scalar *array,Vec *x);
VecCreateMPIWithArray(MPI_Comm comm,int m,int M,Scalar *array,Vec *x);

Заметим, что здесь обязательно нужно обеспечить значение *m*, оно не может быть задано установкой параметра PETSC_DECIDE, и пользователь несет ответственность за обеспечение достаточного объема в массиве $m \cdot sizeof(Scalar)$.

9.2. ОСНОВНЫЕ ВЕКТОРНЫЕ ОПЕРАЦИИ

В табл. 9.1 представлены векторные операции, поддерживаемые PETSc. Эти операции были выбраны потому, что они часто встречаются в приложениях.

Таблица.9.1

Основные векторные операции в PETSc

Название функции	Операция
VecAXPY(Scalar *a,Vec x, Vec y);	$y = y + ax$
VecAYPX(Scalar *a,Vec x, Vec y);	$y = x + ay$
VecWAXPY(Scalar *a,Vec x,Vec y, Vec w);	$w = ax + y$
VecAXPBY(Scalar *a,Scalar *,Vec x,Vec y);	$y = ax + by$
VecScale(Scalar *a, Vec x);	$x = ax$
VecDot(Vec x, Vec y, Scalar *r);	$r = \bar{x}'y$
VecTDot(Vec x, Vec y, Scalar *r);	$r = x'y$
VecNorm(Vec x,NormType type, double *r);	$r = \|x\|_{type}$
VecSum(Vec x, Scalar *r);	$r = \sum x_i$
VecCopy(Vec x, Vec y);	$y = x$
VecSwap(Vec x, Vec y);	$y = x\, while\, x = y$
VecPointwiseMult(Vec x,Vec y, Vec w);	$w_i = x_i y_i$
VecPointwiseDivide(Vec x,Vec y, Vec w);	$w_i = x_i / y_i$
VecMDot(int n,Vec x, Vec *y,Scalar *r);	$r[i] = \bar{x}'y[i]$
VecMTDot(int n,Vec x, Vec *y,Scalar *r);	$r[i] = x'y[i]$
VecMAXPY(int n, Scalar *a,Vec y, Vec *x);	$y = y + \sum_i a_i x[i]$
VecMax(Vec x, int *idx, double *r);	$r = \max x_i$
VecMin(Vec x, int *idx, double *r);	$r = \min x_i$
VecAbs(Vec x);	$x_i = \|x_i\|$
VecReciprocal(Vec x);	$x_i = 1/x_i$
VecShift(Scalar *s,Vec x);	$x_i = s + x_i$

Для параллельных векторов, распределенных по процессам, локальный номер процесса можно определить процедурой

VecGetOwnershipRange(Vec vec,int *low,int *high);

Аргумент *low* указывает на первый компонент, принадлежащий локальному процессу, а аргумент *high* показывает на единицу больший, чем последний, принадлежащий локальному процессу. Эта команда полезна, например, при сборке параллельных векторов.

В случае, когда пользователю нужно выбрать реальный элемент вектора, процедура VecGetArray() возвращает указатель на элементы, локальные для процесса:

VecGetArray(Vec v,Scalar **array);

Когда доступ к массиву больше не нужен, пользователь вызывает

VecRestoreArray(Vec v, Scalar **array);

Заметим, что процедуры VecGetArray() и VecRestoreArray() не копируют векторные элементы, а дают пользователю прямой доступ к векторным элементам. Поэтому эти процедуры по существу не требуют времени для выполнения вызова и являются эффективными. Количество элементов, хранимых локально, может быть получено процедурой

VecGetLocalSize(Vec v,int *size);

Длина глобального вектора определяется процедурой

VecGetSize(Vec v,int *size);

Дополнительно к процедурам VecDot(), VecMDot(),VecNorm() PETSc имеет версии с расщеплением фазы этих операций, что позволяет нескольким независимым внутренним произведениям и/или нормам разделять один и тот же коммуникатор (тем самым, повышая эффективность параллельных вычислений). Например, пусть написан следующий код:

```
VecDot(Vec x,Vec y,Scalar *dot);
VecNorm(Vec x,NormType NORM_2,double *norm2);
VecNorm(Vec x,NormType NORM_1,double *norm1);
```

Этот код работает хорошо, но проблема состоит в том, что выполняются три операции обмена. Вместо этого можно написать

```
VecDotBegin(Vec x,Vec y,Scalar *dot);
VecNormBegin(Vec x,NormType NORM_2,double *norm2);
VecNormBegin(Vec x,NormType NORM_1,double *norm1);
```

VecDotEnd(Vec x,Vec y,Scalar *dot);
VecNormEnd(Vec x,NormType NORM_2,double *norm2);
VecNormEnd(Vec x,NormType NORM_1,double *norm1);

По этому коду обмен задерживается до первого обращения к процедуре VecxxxEnd(), для которой требуется единственная редукция, чтобы совершить обмен всех требуемых значений. Необходимо, чтобы обращения к процедуре VecxxxEnd() выполнялись в том же порядке, как и обращения к VecxxxBegin(). Однако если ошибочно сделано обращение не в правильном порядке, PETSc будет генерировать ошибку, информируя об этом.

Две процедуры VecTDotBegin() и VecTDotEnd() работают только в MPI-1, следовательно, они не позволяют совмещать вычисления с обменом. Поскольку MPI-2 является более общим инструментом, эти процедуры будут совершенствоваться, чтобы совмещать вычисление внутреннего произведения и вычисление нормы с другими вычислениями.

9.3. ИНДЕКСАЦИЯ И УПОРЯДОЧИВАНИЕ

Написанию параллельных программ для решения ДУЧП сопутствует повышенная сложность, вызываемая наличием множественных путей индексации (перечисления) и упорядочивания объектов, таких, как узлы. Например, генератор сеток или разделитель может перенумеровывать вершины, что приводит к корректировке других структур данных, которые ссылаются на эти объекты.

Кроме того, локальная нумерация (на одном процессе) может отличаться от глобальной (межпроцессной). PETSc предоставляет средства, которые помогают управлять разметкой между различными системами нумерации. Основными являются: AO (**A**pplication **O**rdering) – прикладное упорядочивание, которое устанавливает соответствие между различными глобальными (межпроцессными) схемами нумерации; и ISLocalToGlobal Mapping, которое устанавливает соответствие между локальными и глобальными нумерациями.

9.3.1. Прикладное упорядочивание

Во многих приложениях желательно работать с несколькими системами упорядочивания или нумерации степеней свободы клеток, узлов и др. Но в параллельной среде это сделать сложно, поскольку каждый процесс не может хранить полные списки соответствия между различными нумерациями.

Кроме того, упорядочивания, используемые в процедурах линейной алгебры PETSc, могут не соответствовать естественному упорядочиванию для приложения. PETSc содержит полезные процедуры, которые позволяют ясно и эффективно работать с различными упорядочиваниями. Чтобы определить новое прикладное упорядочивание (AO в PETSc), нужно вызвать процедуру

```
AOCreateBasic(MPI_Comm comm,int n,int *apordering,
              int *petscordering,AO *ao);
```

Массивы упорядочивания *apordering* и *petscordering* соответственно содержат список целых чисел в прикладном упорядочивании и их соответствующие значения в PETSc упорядочивании. Каждый процесс может обеспечить любое подмножество упорядочивания, которое он выбрал, но множественные процессы никогда не должны создавать дублированные значения.

Например, рассмотрим вектор длины 5, где узел 0 в прикладном упорядочивании соответствует узлу 3 в PETSc упорядочивании. Дополнительно узлы 1, 2, 3 и 4 прикладного упорядочивания соответствуют узлам 2, 1, 4 и 0 PETSc-упорядочивания. Мы можем записать это соответствие как

$$0, 1, 2, 3, 4 \rightarrow 3, 2, 1, 4, 0.$$

Пользователь может создать AO-соответствие в PETSc несколькими путями. Например, если используются два процесса, то можно вызвать

```
AOCreateBasic(PETSC_COMM_WORLD,2,{0,3},{3,4},&ao);
```

на первый процесс и

```
AOCreateBasic(PETSC_COMM_WORLD,3,{1,2,4},{2,1,0},&ao);
```

на второй процесс. Когда прикладное упорядочивание создано, оно может использоваться любой из команд

```
AOPetscToApplication(AO ao,int n,int *indices);
AOApplicationToPetsc(AO ao,int n,int *indices);
```

На входе *n*-размерный массив *indices* описывает индексы, подлежащие согласованию, а на выходе содержит согласованные значения. Поскольку в общем применяются параллельные базы данных для AO-согласования, критично, чтобы все процессы, которые вызывают AOCreateBasic(), также вызывали эти процедуры, они не могут быть вызваны только подмножеством процессов в коммуникаторе, который был использован в вызове AOCreateBasic().

Процедурой для создания прикладного упорядочивания AO является процедура

AOCreateBasicIS(IS apordering,IS petscordering,AO *ao);

где вместо целочисленных массивов использованы индексные ряды.

Согласующие процедуры

AOPetscToApplicationIS(AO ao,IS indices);
AOApplicationToPetscIS(AO ao,IS indices);

будут согласовывать индексные ряды (IS objects) между упорядочиваниями. Как AOXxxToYyy(), так и AOXxxToYyyIS() могут быть использованы в зависимости от того, был ли AO создан с помощью процедуры AOCreateBasic() или процедуры AOCreateBasicIS(). AO-контекст может быть удален с помощью

AODestroy(AO ao);

и визуализован с помощью процедуры

AOView(AOao,PetscViewerviewer).

Процедура AOxxToxx() позволяет использовать отрицательные элементы во входных целочисленных массивах. Эти элементы не согласовываются, они просто остаются неизменными. Это позволяет, например, согласовывая списки смежности, которые не используют отрицательные числа, отмечать несуществующих соседей.

9.3.2. Локально-глобальное соответствие

Во многих приложениях работают с глобальным представлением вектора (обычно полученным с помощью процедуры VecCreateMPI()) и локальным представлением того же самого вектора, который включает теневые точки, требуемые для локальных вычислений. PETSc имеет процедуры для преобразования индексов от локальной схемы нумерации к глобальной. Это делается следующими процедурами:

```
ISLocalToGlobalMappingCreate(int N,int* globalnum,
                             ISLocalToGlobalMapping* ctx);
ISLocalToGlobalMappingApply(ISLocalToGlobalMapping ctx,
                            int n,int *in,int *out);
ISLocalToGlobalMappingApplyIS(ISLocalToGlobalMapping ctx,
                              IS in,IS* isout);
ISLocalToGlobalMappingDestroy(ISLocalToGlobalMapping ctx);
```

Здесь *N* обозначает число локальных индексов, *globalnum* содержит глобальный номер каждого локального номера, и *ISLocalToGlobal-*

Mapping есть результирующий PETSc объект, который содержит информацию, необходимую для применения ISLocalToGlobalMappingApply или ISLocalToGlobal-Mapping ApplyIS.

Процедуры ISLocalToGlobalMapping служат другим целям, чем AO-процедуры. Они обеспечивают преобразование от локальной нумерации к глобальной, в то время как в случае AO обеспечивается соответствие между двумя глобальными схемами нумерации.

Фактически могут применять как AO, так и процедуры ISLocalToGlobalMapping. Процедуры AO используются первыми для перехода от прикладного глобального упорядочивания (которое не имеет никакого отношения к параллельной обработке) к схеме PETSc (где каждый процесс имеет длинный ряд индексов в нумерации).

Тогда, чтобы вычислить функцию или оценить якобиан локально на каждом процессе, работают с локальной схемой нумерации, которая включает теневые точки. Переход от этой локальной схемы обратно к глобальной PETSc-нумерации может быть выполнен процедурой ISLocalToGlobalMapping. Если имеется список индексов в глобальной нумерации, то процедура

```
ISGlobalToLocalMappingApply(ISLocalToGlobalMapping ctx,
                            ISGlobalToLocalMappingType type,int nin,
                            int *idxin,int *nout,int *idxout);
```

будет создавать новый список индексов в локальной нумерации. Отрицательные значения в *idxin* остаются несогласованными. Однако если *type* принимает значение IS_GTOLM_MASK, *nout* – значение *nin*, то все глобальные значения в *idxin*, которые не представлены в локально-глобальном соответствии, замещаются на –1. Когда тип установлен в IS_GTOLM_DROP, значения в *idxin*, которые не представлены локально в согласовании, не включаются в *idxout*, так что потенциально *nout* меньше, чем *nin*.

Необходимо установить размер массива, чтобы разместить все индексы. Для этого можно вызвать процедуру ISGlobalToLocal-MappingApply() с *idxout*, равным PETSC_NULL, чтобы определить требуемую длину (возвращается в *nout*), и затем распределить требуемый объем и вызвать эту процедуру второй раз, чтобы установить значения.

Часто удобно установить значения вектора, используя локальную нумерацию узлов, а не глобальную (т. е. процесс может содержать его собственный подсписок узлов и элементов и нумеровать их локально).

Чтобы установить значения вектора с локальной нумерацией, необходимо вызвать

VecSetLocalToGlobalMapping(Vecv,ISLocalToGlobalMapping ctx);
VecSetValuesLocal(Vec x,int n,int *indices,Scalar *values,INSERT_VALUES);

Теперь индексы используют локальную, а не глобальную нумерацию.

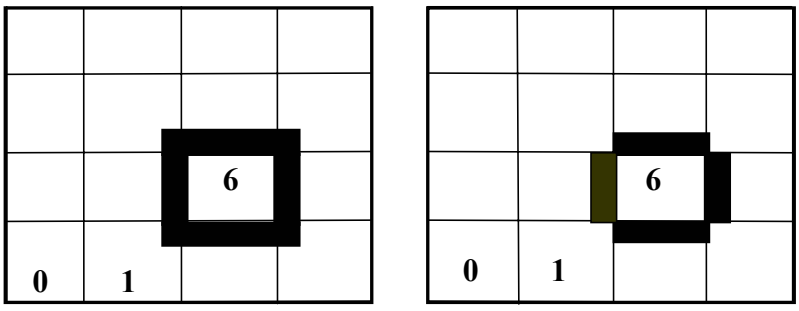
9.4. СТРУКТУИРОВАННЫЕ СЕТКИ, ИСПОЛЬЗУЮЩИЕ РАСПРЕДЕЛЕННЫЕ МАССИВЫ

Распределенные массивы **DA** (**D**istributed **A**rrays), которые используются совместно с векторами PETSc, предназначены для работы с логическими регулярными прямоугольными сетками, когда необходим обмен нелокальными данными перед некоторыми локальными вычислениями.

Эти массивы спроектированы только для случая, когда данные можно представить хранящимися в многомерном стандартном массиве, следовательно, DA предназначены для обмена векторной информации и не предназначены для хранения матриц.

Например, типичная ситуация, которая встречается при параллельном решении ДУЧП, состоит в том, что для оценки локальной функции каждому процессу требуется локальная часть вектора вместе с теневыми точками (граничные части вектора, принадлежащие соседним процессам).

На рис. 9.1 представлены теневые точки для процесса с номером *6* в двухмерной регулярной параллельной сетке. Каждый прямоугольник представляет процесс, теневые точки показаны темными.



Шаблон типа ящик Шаблон типа звезда

Рис 9.1. Теневые области для двух типов шаблона для процесса с номером *6*

9.4.1. Создание распределенных массивов

PETSc DA-объект управляет параллельными обменами при работе с данными, хранимыми в регулярных массивах. Разные данные хранятся в соответствующего размера векторных объектах. DA-объекты содержат только топологическую и коммуникационную информацию.

Структура обмена данными в распределенных массивах для двух измерений создается командой

```
DACreate2d(MPI_Comm comm,DAPeriodicType wrap, DAStencilType st,
           int M, int N,int m,int n, int dof,int s,int *lx,int *ly,DA *da);
```

Аргументы *M* и *N* указывают глобальное количество точек сетки в каждом направлении, а *m* и *n* обозначают распределение по процессам в каждом направлении. Произведение *mn* соответствует количеству процессов в коммуникаторе comm.

Вместо описания топологии процессов можно использовать установку PETSC_DECIDE для *m* и *n*, тогда топологию будет определять PETSc, используя MPI. Тип периодичности массива описывается с помощью переменной *wrap*, которая может иметь разные значения: DA_NONPERIODIC (нет периодичности); DA_XYPERIODIC (периодичность в обоих *x* и *y* направлениях); DA_XPERIODIC или DA_YPERIODIC. Аргумент *dof* указывает число степеней свободы на каждой точке массива, а *s* есть ширина шаблона (т. е. ширина области теневых точек). Массивы *lx* и *ly* могут хранить количество узлов вдоль осей *x* и *y* для каждой клетки, т. е. размерность *lx* есть *m*, а размерность *ly* – это *n*; также может быть передан PETSC_NULL.

Для обмена данными в распределенных массивах могут быть созданы два типа структур, как указано в аргументе *st*. Шаблоны звездного типа, которые определяют распространение только в координатных направлениях, задаются при помощи DA_STENCIL_STAR, прямоугольные шаблоны – с помощью DA_STENCIL_BOX.

Например, для двухмерного случая DA_STENCIL_STAR с шириной единица соответствует стандартному 5-точечному шаблону, а DA_STENCIL_BOX с шириной единица – стандартному 9-точечному шаблону. В обоих примерах теневые точки идентичны, единственная разница состоит в том, что в звездном шаблоне определенные теневые точки игнорируются, потенциально уменьшая число посланных сообщений. Заметим, что шаблон DA_STENCIL_STAR может уменьшить межпроцессные обмены в двух и трех измерениях.

Команды создания коммуникационных структур данных для распределенных массивов для одного и трех измерений аналогичны:

```
DACreate1d(MPI_Comm comm,DAPeriodicType wrap, int M,
           int w, ints, int *lc, DA *inra);

DACreate3d(MPI_Comm comm,DAPeriodicType wrap,
           DAStencilType stencil_type, int M, int N,int P,int m,int n,
           int p, int w, int s, int *lx, int *ly, int *lz, DA *inra);
```

Дополнительными опциями в трех измерениях для *DAPeriodicType* являются DA_ZPERIODIC, DA_XZPERIODIC, DA_YZPERIODIC и DA_XYZPERIODIC. Процедуры для создания распределенных массивов являются коллективными, так что все процессы в коммуникаторе comm обязаны вызвать процедуру DACreateXxx().

9.4.2. Локально/глобальные векторы и распределение данных

Каждый распределенный DA-объект определяет размещение двух векторов: распределенного глобального вектора и локального вектора, который включает место и для теневых точек. DA-объекты обеспечивают информацию о размерах и размещении этих векторов. Пользователь может создавать векторные объекты, которые используют информацию о размещении с помощью процедур:

```
DACreateGlobalVector(DA da,Vec *g);
DACreateLocalVector(DA da,Vec *l);
```

Эти векторы будут служить строительными блоками для локального и глобального решения дифференциальных уравнений в частных производных и др. Если необходимы дополнительные векторы с таким размещением информации, они могут быть получены дублированием процедурами VecDuplicate() или VecDuplicateVecs().

Распределенные массивы обеспечивают информацию, необходимую для обмена теневыми значениями между процессами. В большинстве случаев различные векторы могут разделять одну и ту же коммуникационную информацию (другими словами, разделять данный DA). Проектирование DA-объекта делает это легким, так как DA-операции могут выполняться на векторах соответствующего размера, полученного через DACreateLocalVector() и DACreateGlobalVector() или произведенного процедурой VecDuplicate(). Если это так, то DA-операции scatter/gather (т. е. DAGlobalToLocalBegin()) требуют input/output аргументов.

PETSc в настоящее время не имеет контейнера для множества массивов, разделяющих те же коммуникации для распределенных массивов. На определенной ступени многих приложений необходимо работать с локальной частью вектора, включающей теневые точки. Это может быть сделано рассылкой глобального вектора в его локальные части использованием двухступенчатых команд:

DAGlobalToLocalBegin(DA da,Vec g,InsertMode iora,Vec l);
DAGlobalToLocalEnd(DA da,Vec g,InsertMode iora,Vec l);

которые позволяют совмещать вычисления с обменом. Поскольку локальные и глобальные векторы обязаны совмещаться с распределенным массивом DA, они должны генерироваться через процедуры DACreateGlobalVector() и DACreateLocalVector() (или быть дубликатами такого вектора, полученными через VecDuplicate()).

Аргумент InsertMode может иметь значения ADD_VALUES или INSERT_VALUES. Различные локальные добавки можно рассылать в распределенные вектора командой

DALocalToGlobal(DA da,Vec l,InsertMode mode,Vec g);

Заметим, что эта функция не подразделяется на начальную и конечную фазы, поскольку она чисто локальная.

Третий тип рассылки распределенных массивов состоит из рассылки из локального вектора (включая теневые точки, которые содержат произвольные значения) в локальный вектор с правильными значениями теневых точек. Эта рассылка может быть сделана следующим образом:

DALocalToLocalBegin(DA da,Vec l1,InsertMode iora,Vec l2);
DALocalToLocalEnd(DA da,Vec l1,InsertMode iora,Vec l2);

Поскольку оба локальных вектора *l*1 и *l*2 обязаны быть совместимыми с распределенным массивом DA, они получаются процедурой DACreateLocalVector() (или являются дубликатом таких векторов, полученными через VecDuplicate()). Значениями InsertMode могут быть ADD_VALUES или INSERT_VALUES. К контексту векторной рассылки можно обращаться прямо, используя в local-to-global (ltog), global-to-local (gtol) и local-to-local (ltol) рассылках команду

DAGetScatter(DA da,VecScatter *ltog,VecScatter *gtol,VecScatter *ltol);

9.4.3. Локальные (с теневыми точками) рабочие векторы

В большинстве приложений локальные (с теневыми точками) векторы нужны только на этапе вычисления пользователем значения

функций. PETSc обеспечивает легкий способ (без существенных потерь времени) получения этих рабочих векторов и их возврата, когда они больше не нужны. Это делается с помощью процедур

```
DAGetLocalVector(DA da,Vec *l);
.... use the local vector l
DARestoreLocalVector(DA da,Vec *l);
```

9.4.4. Доступ к векторным элементам для DA векторов

PETSc обеспечивает простой способ, чтобы установить значения в DA векторах и получить доступ к ним, используя естественную индексацию сеток. Это осуществляется с помощью вызовов следующих процедур:

```
DAVecGetArray(DA da,Vec l,(void**)array);
... use the array indexing it with 1 or 2 or 3 dimensions
... depending on the dimension of the DA
DAVecRestoreArray(DA da,Vec l,(void**)array);
```

Вектор *l* может быть глобальным или локальным. Доступ в массив осуществляется через глобальную индексацию на полной сетке. Например, для решения скалярной задачи в двух измерениях можно сделать следующее:

```
Scalar **f,**u;
DAVecGetArray(DA da,Vec local,(void**)u);
DAVecGetArray(DA da,Vec global,(void**)f);
f[i][j] = u[i][j] - ...
DAVecRestoreArray(DA da,Vec local,(void**)u);
DAVecRestoreArray(DA da,Vec global,(void**)f);
```

По адресу *${PETSC_DIR}/src/snes/examples/tutorials/ex5.c* размещен законченный пример. По адресу *${PETSC_DIR}/src/snes/examples/tutorials/ex19.c* находится пример для многокомпонентного ДУЧП.

9.4.5. Информация о сетке

Глобальные индексы левого нижнего угла локальной части массива могут быть получены командами

```
DAGetCorners(DA da,int *x,int *y,int *z,int *m,int *n,int *p);
DAGetGhostCorners(DA da,int *x,int *y,int *z,int *m, int *n,int *p);
```

Первая версия исключает любые теневые точки, а вторая – включает их. Когда используется любой тип (DA_STENCIL_STAR или DA_STENCIL_BOX) шаблона, локальные векторы (с теневыми точ-

ками) представляют прямоугольный массив, включая в случае DA_STENC IL_STAR дополнительные угловые элементы. Эта конфигурация обеспечивает простой доступ к элементам путем использования двух- или трехмерной индексации. Единственная разница между этими двумя случаями состоит в том, что при использованиипроцедуры DA_STENCIL_STAR дополнительные угловые компоненты не рассылаются между процессами и поэтому содержат неопределенные значения, которые не следует использовать.

Чтобы собрать глобальную плотную матрицу, можно поступить двумя способами. В первом случае можно определить глобальный номер узла для каждого локального узла командой

DAGetGlobalIndices(DA da,int *n,int **idx).

Выходной аргумент *n* содержит количество локальных узлов, включая теневые, а параметр *idx* содержит список глобальных индексов, который соответствует локальным узлам.

Во втором случае установим векторы и матрицы так, чтобы их элементы можно было добавить с использованием локальной нумерации. Для этого нужно осуществить вызов следующих процедур:

DAGetISLocalToGlobalMapping(DA da,ISLocalToGlobalMapping *map);
VecSetLocalToGlobalMapping(Vec x,ISLocalToGlobalMapping map);
MatSetLocalToGlobalMapping(Vec x,ISLocalToGlobalMapping map);

Теперь элементы могут быть прибавлены к векторам и матрицам с помощью локальной нумерации и функций VecSetValuesLocal() и MatSetValuesLocal().

Поскольку глобальное упорядочивание, которое PETSc использует для управления параллельными векторами (и матрицами), обычно не соответствует «натуральному» упорядочиванию на двух- и трехмерных массивах, DA-структура обеспечивает прикладное упорядочивание AO, которое устанавливает соответствие между естественным упорядочиванием на прямоугольной сетке и упорядочиванием, которое PETSc использует для параллельной работы. Этот контекст упорядочивания можно получить командой

DAGetAO(DA da,AO *ao);

Рис. 9.2 представляет упорядочивание для двухмерного распределенного массива, разделенного на четыре процесса.

Пример *${PETSC_DIR}/src/snes/examples/tutorials/ex5.c* иллюстрирует использование распределенных массивов для решения нелинейных проблем.

Рис. 9.2. Естественное и PETSc упорядочивание для 2D массива (4 процесса)

9.5. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ДЛЯ УПРАВЛЕНИЯ ВЕКТОРАМИ, СВЯЗАННЫМИ С НЕСТРУКТУИРОВАННЫМИ СЕТКАМИ

9.5.1. Индексные ряды

Чтобы облегчить векторную рассылку и сборку (например, при обновлении теневых точек) для задач, определенных на неструктуированных сетках, PETSc применяет концепцию индексных рядов. Индексный ряд, который является обобщением ряда целочисленных индексов, используется, чтобы определить рассылку, сборку и подобные операции на векторах и матрицах.

Следующие команды создают индексный ряд, основанный на списке целых чисел:

ISCreateGeneral(MPI_Comm comm,int n,int *indices, IS *is);

Эта процедура по существу копирует *n* индексов, переданных ей целочисленным массивом *indices*. Другой стандартный индексный ряд определен стартовой точкой (*first*) и страйдом (шагом) и может быть создан командой

ISCreateStride(MPI_Comm comm,int n,int first,int step,IS *is);

Индексный ряд может быть удален командой

ISDestroy(IS is);

В некоторых случаях пользователь может иметь доступ к информации прямо из индексного ряда. В этом помогает несколько команд:

ISGetSize(IS is,int *size);
ISStrideGetInfo(IS is,int *first,int *stride);
ISGetIndices(IS is,int **indices);

Функция ISGetIndices() возвращает указатель на список индексов в индексном ряду. Для определенных индексных рядов это может быть временный массив индексов, создаваемый специально для данной процедуры. Поэтому, как только пользователь закончил использовать массив индексов, должна быть вызвана процедура

ISRestoreIndices(IS is, int **indices);

чтобы гарантировать освобождение пространства, которое система могла использовать для генерации списка индексов. Блокирующая версия индексных рядов может быть создана командой

ISCreateBlock(MPI_Comm comm,int bs,int n,int *indices, IS *is);

Эта версия используется для определения операций, в которых каждый элемент индексного ряда ссылается на блок, состоящий из *bs* векторных элементов. Существуют процедуры, аналогичные вышеприведенным: ISBlockGetIndices(), ISBlockGetSize(), ISBlock(), и ISBlockGetBlockSize().

9.5.2. Рассылки и сборки

Векторы PETSc имеют полную поддержку для выполнения операций рассылки и сборки. Можно выбрать любое подмножество компонент вектора, чтобы вставить или прибавить его к любому подмножеству компонент другого вектора. Рассмотрим эти операции как обобщение рассылок, хотя в действительности они являются комбинацией рассылки и сборки.

Чтобы скопировать избранные компоненты из одного вектора в другой, используются следующие команды:

VecScatterCreate(Vec x,IS ix,Vec y,IS iy,VecScatter *ctx);
VecScatterBegin(Vecx,Vecy,INSERT_VALUES,SCATTER_FORWARD,
VecScatter ctx);
VecScatterEnd(Vec x,Vec y,INSERT_VALUES,SCATTER_FORWARD,
VecScatter ctx);
VecScatterDestroy(VecScatter ctx);

Здесь *ix* обозначает индексный ряд первого вектора, а *iy* – индексный ряд вектора назначения. Векторы могут быть последовательными или параллельными. Требований немного – число элементов индексного ряда первого вектора *ix* должно равняться числу элементов индексно-

го ряда второго вектора *iy*. Кроме того, индексные ряды должны быть достаточно длинными, чтобы содержать все индексы, на которые есть ссылка. Аргумент *INSERT_VALUES* указывает, что векторные элементы вставлены в указанные ячейки вектора назначения, переписывая существующие значения. Чтобы добавить компоненты, а не вставить их, пользователю нужно выбрать опцию *ADD_VALUES* вместо *INSERT_VALUES*.

Чтобы выполнить обычную операцию сборки, пользователь просто создает индексный ряд назначения *iy* со страйдом единица. Аналогично традиционная рассылка может быть выполнена с начальным (посылающим) индексным рядом, содержащим страйд. Для параллельных векторов все процессы, которые имеют векторы, обязаны вызывать процедуры рассылки.

Когда рассылка производится от параллельного вектора к последовательному, каждый процесс имеет собственный последовательный вектор, который принимает значения из ячеек, как указано в его собственном индексном ряду. Аналогично, когда рассылка производится от последовательного вектора к параллельному, каждый процесс имеет его собственный последовательный вектор, который делает вклад в параллельный вектор. Если используется *INSERT_VALUES* и два различных процесса вносят два различных значения в один и тот же компонент параллельного вектора, может оказаться вставленным любое значение. В некоторых случаях может оказаться необходимым отменить рассылку, то есть выполнить обратную рассылку, поменяв роли посылающего и принимающего процессов. Это выполняется использованием процедур

```
VecScatterBegin(Vec y,Vec x,INSERT_VALUES,
                    SCATTER_REVERSE,VecScatter ctx);
VecScatterEnd(Vec y,Vec x,INSERT_VALUES,
                  SCATTER_REVERSE,VecScatter ctx);
```

Когда VecScatter создан, он может быть использован с любым вектором, который имеет соответствующее для параллельных данных размещение. Это означает, что можно вызвать процедуры VecScatterBegin() и VecScatterEnd() с другими векторами, чем те, которые использовались в процедуре VecScatterCreate(), до тех пор, пока они имеют то же самое параллельное размещение (то же самое число элементов на каждом процессе). Обычно эти «различные» векторы получаются через обращение к VecDuplicate() от исходных векторов, использованных при вызове VecScatterCreate().

В PETSc не имеется процедур, противоположных VecSetValues(), например, таких, как VecGetValues(). Вместо этого пользователю следует создать новый вектор, где должны храниться компоненты, и выполнить соответствующую векторную рассылку. Например, если нужно получить значения 100-го и 200-го элементов параллельного вектора *p*, можно использовать программу, похожую на ту, которая представлена на рис. 9.3.

```
Vec       p, x;              /* initial vector, destination vector */
VecScatter scatter;          /* scatter context */
IS        from, to;          /* index sets that define the scatter */
Scalar    *values;
int       idx_from[] = {100,200}, idx_to[] = {0,1};

VecCreateSeq(PETSC_COMM_SELF,2,&x);
ISCreateGeneral(PETSC_COMM_SELF,2,idx_from,&from);
ISCreateGeneral(PETSC_COMM_SELF,2,idx_to,&to);
VecScatterCreate(p,from,x,to,&scatter);
VecScatterBegin(p,x,INSERT_VALUES,SCATTER_FORWARD,scatter);
VecScatterEnd(p,x,INSERT_VALUES,SCATTER_FORWARD,scatter);
VecGetArray(x,&values);
ISDestroy(from);
ISDestroy(to);
VecScatterDestroy(scatter);
```

Рис. 9.3. Пример программы для векторной рассылки

В этом примере значения указанных компонент размещаются в массиве *values*. Каждый процесс теперь имеет 100-ю и 200-ю компоненту, но очевидно, что каждый процесс мог бы собрать любые необходимые элементы созданием индексного ряда без элементов. Рассылка содержит две ступени, это позволяет совместить обмен и вычисления. Введение контекста для VecScatter позволяет повторно использовать однажды вычисленные для рассылки шаблоны обмена. Установка обмена для рассылки требует обмена, поэтому лучше, где это возможно, повторно использовать такую же информацию.

9.5.3. Рассылка теневых значений

Рассылка представляет общий метод для управления требуемыми теневыми значениями для неструктуированных сетевых вычислений. Пусть производится рассылка глобального вектора в локальный «теневой» рабочий вектор, выполняются вычисления на локальных рабочих векторах, затем выполняется рассылка обратно в глобальный век-

тор решения. В простейшем случае это может быть записано следующим образом:

```
Function:  (Input Vec globalin, Output Vec globalout)
VecScatterBegin(Vec globalin,Vec localin,InsertMode INSERT_VALUES,
                    ScatterMode SCATTER_FORWARD,VecScatter scatter);
VecScatterEnd(Vec globalin,Vec localin,InsertMode INSERT_VALUES,
                   ScatterMode SCATTER_FORWARD,VecScatter scatter);

    /*    For example, do local calculations from localin to localout  */
VecScatterBegin(Vec localout,Vec globalout,InsertMode ADD_VALUES,
                    ScatterMode SCATTER_REVERSE,VecScatter scatter);
VecScatterEnd(Vec localout,Vec globalout,InsertMode ADD_VALUES,
                  ScatterMode SCATTER_REVERSE,VecScatter scatter);
```

9.5.4. Векторы с ячейками для теневых значений

Параграф содержит информацию о более эффективном использовании PETSc-векторов. В описанном выше основном подходе имеется два небольших препятствия.

- Высоки требования к памяти для вектора *localin*, который дублирует память в *globalin.*
- Для копирования локальных значений из *localin* в *globalin* требуется большое время.

Альтернативный подход состоит из размещения глобальных векторов в пространстве, предразмещенном для теневых значений. Это может быть сделано с помощью

```
VecCreateGhost(MPI_Comm comm,int n,int N,int nghost, int *ghosts,Vec *vv);
VecCreateGhostWithArray(MPI_Comm comm,int n,int N,int nghost,
                                int *ghosts,Scalar *array,Vec *vv)
```

Здесь *n* – количество элементов локального вектора; *N* – количество глобальных элементов (может быть равно PETSC_NULL); *nghost* – количество теневых элементов. Массив *ghosts* имеет размер *nghost* и содержит ячейку глобального вектора для каждой локальной теневой ячейки. Использование VecDuplicate() или VecDuplicateVecs() для теневого вектора будет вызывать генерацию дополнительных теневых векторов.

Во многих случаях теневой вектор ведет себя как любой другой MPI-вектор, созданный процедурой VecCreateMPI(). Разница состоит в том, что теневой вектор имеет дополнительное «локальное» представление, которое обеспечивает доступ к теневым ячейкам. Это сделано через обращение к процедуре

VecGhostGetLocalForm(Vec g,Vec *l).

Вектор *l* есть последовательное представление параллельного вектора *g*, который разделяет то же пространство массивов (и, следовательно, числовые значения), однако позволяет обращаться к «теневым» значениям за «концом» массива. Заметим, что к элементу *l* обращаются через локальную нумерацию элементов и теневых точек, в то время как при обращении к *g* – через глобальную нумерацию. Общее использование теневого вектора задается следующим образом:

```
VecGhostUpdateBegin(Vec globalin,InsertMode INSERT_VALUES,
                        ScatterModeSCATTER_FORWARD);
VecGhostUpdateEnd(Vec globalin,InsertMode INSERT_VALUES,
                      ScatterModeSCATTER_FORWARD);
VecGhostGetLocalForm(Vec globalin,Vec *localin);
VecGhostGetLocalForm(Vec globalout,Vec *localout);
                    /* Do local calculations from localin to localout  */
VecGhostRestoreLocalForm(Vec globalin,Vec *localin);
VecGhostRestoreLocalForm(Vec globalout,Vec *localout);
VecGhostUpdateBegin(Vec globalout,InsertMode
                        ADD_VALUES,ScatterModeSCATTER_REVERSE);
VecGhostUpdateEnd(Vec globalout,InsertMode ADD_VALUES,
                        ScatterModeSCATTER_REVERSE);
```

Заметим, что, процедуры VecGhostUpdateBegin/End() и VecScatterBegin/End() эквивалентны за исключением того, что первым, поскольку они рассылают в теневые ячейки, не нужно копировать значения локальных векторов, которые уже есть на месте. Кроме того, пользователь не должен размещать локальный рабочий вектор, поскольку теневой вектор уже имеет размещенные слоты (выделенные места) для хранения теневых значений. Входные аргументы *INSERT_VALUES* и *SCATTER_FORWARD* вызывают коррекцию теневых значений соответствующим процессом. Обновляют «локальные» части вектора, полученные от теневых значений всех других процессов, аргументы *ADD_VALUES* и *SCATTER_REVERSE*

Глава 10. МАТРИЦЫ

10.1. ВВЕДЕНИЕ

PETSc использует различные способы представления матриц, поскольку какой-либо один матричный формат не в состоянии решить все проблемы. В настоящее время поддерживаются форматы для хранение плотных и разреженных матриц, сжатых по строкам (последо-

вательная и параллельная версия), а также несколько специализированных форматов. Могут быть введены и дополнительные форматы.

В главе описываются основы использования матриц в общем (безотносительно к частному выбранному формату), обсуждаются вопросы эффективного использования нескольких простых матричных типов для последовательной и параллельной обработки. Использование PETSc матриц предусматривает следующие действия: создание частных типов матриц, ввод значений в них, обработку матриц, использование матриц для различных вычислений, удаление матриц.

10.2. СОЗДАНИЕ И СБОРКА МАТРИЦ

Простейшая процедура для формирования PETSc матрицы:

MatCreate(MPI_Comm comm,int m,int n,int M,int N,Mat *A);

Эта процедура генерирует последовательную матрицу *A* для выполнения на одном процессе и параллельную матрицу для двух или более процессов. Конкретный формат устанавливается пользователем командами базы опций. Пользователь задает глобальные (*M* и *N*) и локальные (*m* и *n*) размерности матрицы, а PETSc полностью управляет распределением памяти. Эта процедура облегчает переключение между различными типами матриц, например, чтобы определить формат, наиболее удобный для некоторого применения. По умолчанию MatCreate() использует разреженный AIJ формат.

Чтобы вставить или прибавить элементы к матрице, можно вызвать различные варианты процедуры MatSetValues:

MatSetValues(Mat A,int m,int *im,int n,int *in, Scalar *values,
INSERT_VALUES);
MatSetValues(Mat A,int m,int *im,int n,int *in,Scalar *values, ADD_VALUES);

Эта процедура вставляет или добавляет в матрицу логически плотный субблок размера *mn*. Целые индексы *im* и *in* соответственно указывают номера глобальной строки и столбца, куда производится вставка. В процедуре MatSetValues() используется соглашение стандарта языка C, в котором строки и столбцы нумеруются от нуля. Массив *values* является логически двухмерным и содержит вставляемые значения. По умолчанию эти значения расположены по строкам. Чтобы ввести размещения по столбцам, можно вызвать команду

MatSetOption(Mat A,MAT_COLUMN_ORIENTED);

Если используется блочно сжатый формат разреженной по строкам матрицы (MATSEQBAIJ или MATMPIBAIJ), можно вставить элементы более эффективно, используя блочный вариант процедуры:

MatSetValuesBlocked().

Функция MatSetOption() имеет несколько других входов. Подробности можно посмотреть в справочных страницах. Рассмотрим две из этих опций, которые относятся к процессу сборки. Чтобы указать PETSc, какие индексы строк (*im*) и столбцов (*in*) сортируются, можно использовать один из вариантов команды

MatSetOption(Mat A,MAT_ROWS_SORTED);
MatSetOption(Mat A,MAT_COLUMNS_SORTED);

Флаги указывают формат данных, вводимых процедурой MatSetValues(). Они не связаны со способом хранения данных разреженной матрицы внутри PETSc.

После того как элементы вставлены или добавлены в матрицу, они должны быть обработаны перед использованием. Процедуры для обработки матриц:

MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);

Размещая код между этими двумя обращениями, пользователь может выполнить вычисления, пока сообщение находится в состоянии передачи.

Обращения к процедуре MatSetValues() с опциями INSERT_VALUES и ADD_VALUES не могут быть смешаны без включения обращений к процедурам сборки. Для таких промежуточных обращений к сборке второй аргумент процедуры должен быть MAT_FLUSH_ASSEMBLY, который выпускает некоторые работы из полного процесса сборки. Этот аргумент необходим только в последнем ассемблировании матрицы перед ее использованием.

Значения можно вставлять в матрицу вне зависимости от того, какой процесс в настоящий момент их содержит, рекомендуется генерировать большую часть элементов в том процессе, где эти данные будут храниться. В этом случае для матриц, размещенных по процессам, используется процедура

MatGetOwnershipRange(Mat A,int *first_row,int *last_row);

Она информирует пользователя, как все строки от *first_row* до *last_row* – 1 будут храниться в локальном процессе.

В разреженных матричных реализациях после выполнения процедур сборки, матрицы сжимаются и могут быть использованы для матрично-векторного умножения или других операций. Ввод новых значений в матрицу на этой точке может оказаться затратным, поскольку это потребует копирования и, возможно, распределения памяти. Поэтому следует, насколько возможно, устанавливать эти значения перед обращением к конечным процедурам сборки.

Если нужно повторно собрать матрицы, которые остаются тем же ненулевым образцом (например, в нелинейных задачах), должна быть использована опция

MatSetOption(Mat mat,MAT_NO_NEW_NONZERO_LOCATIONS);

после того, как первая матрица была собрана. Эта опция гарантирует, что определенные структуры данных и коммуникационная информация будут повторно использованы (вместо регенерации) на протяжении последующих шагов, обеспечивая повышение эффективности. В файле *${PETSC_DIR}/src/sles/examples/tutorials/ex5.c* содержится простой пример решения двух линейных систем, которые используют одну и ту же структуру данных.

10.2.1. Разреженные матрицы

По умолчанию матрицы в PETSc представлены в общем разреженном AIJ формате. Он называется the Yale sparse matrix format или **C**ompressed **S**parse **R**ow format (CSR). В параграфе обсуждаются некоторые детали эффективного использования этого формата для приложений большого размера. Дополнительные форматы (блочного сжатия по строкам или блочного хранения диагоналей, которые обычно много эффективнее для задач со многими степенями свободы на узел) обсуждаются ниже.

Последовательные AIJ разреженные матрицы. В PETSc в формате AIJ хранятся ненулевые элементы по строкам вместе с сопутствующим массивом соответствующих номеров столбцов и массивом указателей на начало каждой строки. Диагональные элементы матрицы хранятся с остальными ненулевыми элементами (не отдельно).

Чтобы создать последовательную AIJ матрицу A с m строками и n столбцами, используют команду

MatCreateSeqAIJ(PETSC_COMM_SELF,int m,int n,int nz,int *nnz,Mat *A);

где *nz* или *nnz* могут быть использованы для предварительного распределения матричной памяти, если установить *nz* = 0 и *nnz* = PETSC_NULL.

Последовательный и параллельный форматы хранения AIJ матриц, если возможно, используют по умолчанию *I-nodes* (идентичные узлы). Поэтому для повышения эффективности попытаемся добиться повторного использования матричной информации для последовательных строк с той же самой ненулевой структурой. Соответствующими опциями из базы опций являются -mat_aij_no_inode (не использовать *inodes*) и -mat_aij_inode_limit<limit> (установить *inode* границу (*max limit*=5)).

Предварительное распределение памяти для последовательных AIJ разреженных матриц. Динамический процесс распределения новой памяти и копирования из старой памяти в новую требует большого расхода памяти. Поэтому, чтобы получить хорошие характеристики при сборке AIJ матриц, критично предразместить необходимую для разреженных матриц память. Пользователь имеет две возможности сделать это с помощью процедуры MatCreateSeqAIJ(). Можно использовать скаляр *nz*, чтобы описать ожидаемое число ненулевых элементов для каждой строки. Это хорошо, если число ненулевых элементов на строку примерно то же, что и в среднем для матрицы (или как удобный первый шаг для предраспределения). Если кто-то недооценил реальное число ненулевых элементов в строке, тогда на протяжении процесса ассемблирования PETSc будет распределять необходимое дополнительное пространство, и это может замедлить вычисления.

Если строки имеют различное число ненулевых элементов, следует попробовать указать (поближе) точное число элементов для различных строк с помощью массива *nnz* длины *m*, где *m* – это число строк, например:

```
int nnz[m];
nnz[0] = <nonzeros in row 0>
nnz[1] = <nonzeros in row 1>
nnz[m-1] = <nonzeros in row m-1>
```

В таком случае для процесса сборки не надо требовать дополнительного времени для распределения памяти, если *nnz* оценено корректно. Использование массива *nnz* особенно важно, если число ненулевых элементов значительно меняется от строки к строке.

Предварительную оценку количества ненулевых элементов можно выполнить заранее, например, для методов конечных разностей следующим образом.

- Разместить целочисленный массив *nnz*.
- Выполнить цикл для сетки, подсчитывая ожидаемое число ненулевых элементов для строки (строк), связанных с различными сетевыми точками.
- Создать разреженную матрицу с помощью функции MatCreateSeqAIJ() или альтернативной функции.
- Выполнить цикл для сетки, генерируя матричные элементы и внося их в матрицу с помощью MatSetValues().

Для типовых расчетов по методу конечных элементов имеем аналогичную процедуру.

- Разместить целочисленный массив *nnz*.
- Построить цикл для узлов, вычисляющий число соседних узлов, которые определяют количество ненулевых элементов для соответствующей строки матрицы.
- Создать разреженную матрицу процедурой MatCreateSeqAIJ() или альтернативно.
- Сделать цикл по элементам, генерируя матричные элементы и вставляя их в матрицу с помощью процедуры MatSetValues().

Опция -log_info заставляет процедуры MatAssemblyBegin() и MatAssemblyEnd() выводить информацию об успехе предразмещения.

Рассмотрим следующий пример для матричного формата MATSEQ AIJ:

```
MatAssemblyEnd_SeqAIJ:Matrix size 10 X 10; storage space:20 unneeded, 100 used
MatAssemblyEnd_SeqAIJ: Number of mallocs during MatSetValues is 0
```

Первая строка указывает, что пользователь предразместил 120 областей, но только 100 областей были использованы. Вторая строка отмечает, что пользователь предразместил достаточную область, чтобы PETSc не требовал дополнительного размещения (затратная операция). В следующем примере пользователь не предразмещает достаточно пространства, на что указывает факт, что объем mallocs очень большой (плохо для эффективности):

```
MatAssemblyEnd_SeqAIJ:Matrix size 10 X 10; storage space:47 unneeded, 1000 used
MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues is 40000
```

Хотя на первый взгляд такие процедуры определения структуры матрицы заранее могут показаться бесполезными, однако они очень

эффективны, поскольку смягчают необходимость динамического построении структуры данных матрицы, которая может быть очень затратной по врмени.

Параллельные AIJ разреженные матрицы. Параллельные разреженные матрицы с AIJ форматом могут быть созданы командой

```
MatCreateMPIAIJ(MPI_Comm comm,int m,int n,int M,int N,
                int d_nz, *d_nnz, int o_nz,int *o_nnz,Mat *A);
```

Аргумент *A* есть вновь созданная матрица, а аргументы *m*, *n*, *M* и *N* указывают соответственно число локальных и глобальных строк и столбцов. Любые локальные и глобальные параметры могут быть определены в PETSc с помощью PETSC_DECIDE. Матрица хранится с фиксированным числом строк в каждом процессе, задаваемым *m* или определяемым PETSc в случае использования PETSC_DECIDE.

Если PETSC_DECIDE не используется для получения аргументов *m* и *n*, пользователь обязан гарантировать, что они выбраны совместимыми с векторами. Для этого сначала рассматривается произведение матрицы на вектор $y = Ax$. Аргумент *m* в создающей матрицу процедуре MatCreateMPIAIJ() обязан соответствовать локальному размеру, используемому в процедуре VecCreateMPI(), которая создает вектор *y*. Аналогично *n* должно соответствовать тому, которое используется, как локальный размер в процедуре VecCreateMPI для *x*. Пользователь обязан установить аргументы *d_nz* = 0, *o_nz* = 0, *d_nnz* = PETSC_NULL и *o_nnz* = PETSC_NULL для PETSc, чтобы управлять динамическим размещением пространства матричной памяти. Это относится к аргументам *nz* и *nnz* для процедуры MatCreateSeqAIJ(). Эти аргументы описывают ненулевую информацию для диагонали (*d_nz* и *d_nnz*) и внедиагональной (*o_nz* и *o_nnz*) части матрицы.

Для квадратной глобальной матрицы определим долю диагонали каждого процесса как его локальные строки и соответствующие столбцы (квадратная подматрица). Каждая внедиагональная доля составляет остаток матрицы (прямоугольная подматрица). Номер в коммуникаторе MPI определяет абсолютное упорядочевание блоков. Это означает, что процесс с номером 0 в коммуникаторе, заданный для MatCreateMPIAIJ, содержит верхнюю строку матрицы; *i*-й процесс в коммуникаторе содержит *i*-й блок матрицы.

Предраспределение памяти для параллельных AIJ разреженных матриц. Как обсуждалось выше, предраспределение критично для достижения хороших характеристик при сборке матриц, так как это уменьшает количество распределений и требуемых копий.

Приведем пример для трех процессов, чтобы показать, как это может быть сделано для матричного формата MATMPIAIJ. Рассмотрим матрицу 8 × 8, которая разделена по умолчанию следующим образом: три строки матрицы в первом процессе, три – во втором и две – в третьем.

1	2	0	0	3	0	0	4
0	5	6	7	0	0	8	0
9	0	10	11	0	0	12	0
13	0	14	15	16	17	0	0
0	18	0	19	20	21	0	0
0	0	0	22	23	0	24	0
25	26	27	0	0	28	29	0
30	0	0	31	32	33	0	34

Тогда диагональная» подматрица *d* в первом процессе задается так:

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 5 & 6 \\ 9 & 0 & 10 \end{pmatrix},$$

а «внедиагональная» подматрица задается так:

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 4 \\ 7 & 0 & 0 & 8 & 0 \\ 11 & 0 & 0 & 12 & 0 \end{pmatrix}.$$

Для первого процесса можно установить *d_nz* = 2 (поскольку каждая строка имеет два ненулевых элемента) или альтернативно установить для *d_nnz* значение {2, 2, 2}. Аргумент *o_nz* можно установить, равным 2, поскольку каждая строка матрицы имеет два ненулевых элемента, или *o_nnz* можно сделать {2, 2, 2}.Для второго процесса *d* подматрица задается так:

$$\begin{pmatrix} 15 & 16 & 17 \\ 19 & 20 & 21 \\ 22 & 23 & 0 \end{pmatrix}.$$

Поэтому можно установить *d_nz* = 3, поскольку максимальное количество ненулевых элементов в каждой строке равно 3, или альтернативно можно установить *d_nnz* равным {3, 3, 2}, указывая тем, что первые две строки будут иметь три ненулевых элемента, а третья – два.

Соответствующая о подматрица для второго процесса есть

$$\begin{pmatrix} 13 & 0 & 14 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 \\ 0 & 0 & 0 & 24 & 0 \end{pmatrix},$$

поэтому можно установить *o_nz* равным 2 или *o_nnz* равным {2, 1, 1}.

Заметим, что пользователь никогда не работает прямо с параметрами *d* и *o*, кроме случая предраспределения памяти, как указано выше. Итак, пользователю не нужно предраспределять абсолютно правильную сумму пространства. Как только будет дана достаточно близкая оценка, будет получена и высокая эффективность сборки матриц.

Как описано выше, опция -log_info будет выводить информацию о результатах предраспределения во время сборки матрицы. Для матричного формата MATMPIAIJ PETSc она будет также составлять список количества элементов для каждого процесса, которые были сгенерированы на разных процессах. Например, выражения

```
[0]MatAssemblyBegin_MPIAIJ:Number of off processor values 10
[1]MatAssemblyBegin_MPIAIJ:Number of off processor values 7
[2]MatAssemblyBegin_MPIAIJ:Number of off processor values 5
```

указывают, что на процессах было сгенерировано немного значений. С другой стороны, выражения

```
[0]MatAssemblyBegin_MPIAIJ:Number of off processor values 100000
[1]MatAssemblyBegin_MPIAIJ:Number of off processor values 77777
```

говорят, что много значений было сгенерировано на «неверных» процессах. Эта ситуация может быть очень неэффективной, поскольку передача значений в «правильный» процесс очень дорога. В приложениях пользователь должен использовать MatGetOwnershipRange() процедуру, которая позволяет генерировать большинство элементов на нужных процессах.

10.2.2. Плотные матрицы

PETSc использует для плотных матриц как последовательные, так и параллельные форматы, где каждый процесс хранит свои элементы по столбцам в стиле языка Fortran. Чтобы создать последовательную плотную PETSc матрицу A размера $m \times n$, пользователю следует вызвать процедуру

MatCreateSeqDense(PETSC_COMM_SELF,int m,int n,

PetscScalar *data,Mat *A);

Чтобы создать параллельную матрицу A, следует вызвать

MatCreateMPIDense(MPI_Comm comm,int m,int n,int M,int N,
PetscScalar *data,Mat *A)

Аргументы m, n, M и N указывают соответственно число локальных и глобальных строк и столбцов. Любые локальные и глобальные параметры могут быть определены PETSc с помощью PETSC_DECIDE. Матрица хранится с фиксированным количеством строк на каждом процессе, задаваемым m, или определяется PETSc в случае PETSC_DECIDE.

В настоящее время PETSc не поддерживает параллельные прямые методы для плотных матриц.

10.3. ОСНОВНЫЕ МАТРИЧНЫЕ ОПЕРАЦИИ

В табл. 10.1 описаны основные матричные операции PETSc. Параллельную матрицу можно умножить на вектор с n локальными элементами, возвращая вектор с m локальными элементами с помощью процедуры

MatMult(Mat A,Vec x,Vec y);

Векторы x и y следует создать с помощью процедур

VecCreateMPI(MPI_Comm comm,n,N,&x);
VecCreateMPI(MPI_Comm comm,m,M,&y);

Если пользователь предлагает PETSc решать, какое количество элементов должно храниться локально (передавая PETSC_DECIDE вторым аргументом в процедуре VecCreateMPI() или, используя процедуру VecCreate()), векторы и матрицы одного размера являются автоматически совместимыми для матрично-векторных операций.

Кроме процедур умножения, имеется версия для транспонирования матрицы

MatMultTranspose(Mat A,Vec x,Vec y);

Есть версии, которые прибавляют результат к другому вектору:

MatMultAdd(Mat A,Vec x,Vec y,Vec w);
MatMultTransAdd(Mat A,Vec x,Vec y,Vec w);

Эти процедуры соответственно выполняют операции $w = Ax + y$ и $w = A^T x + y$. В языке С векторы y и w могут быть идентичными.

Таблица 10.1

Основные матричные операции

Название функции	Операция
MatAXPY(Scalar *a,Mat X, Mat Y);	$Y = Y + aX$
MatMult(Mat A,Vec x, Vec y);	$y = Ax$
MatMultAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + Ax$
MatMultTrans(Mat A,Vec x, Vec y);	$y = A^T x$
MatMultTransAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T x$
MatNorm(Mat A,NormType type, double *r);	$r = \|A\|_{type}$
MatDiagonalScale(Mat A,Vec l,Vec r);	$A = diag(l) \cdot A \cdot diag(r)$
MatScale(Scalar *a,Mat A);	$A = aA$
MatConvert(Mat A,MatType type,Mat *B);	$B = A$
MatCopy(Mat A,Mat B,MatStructure);	$B = A$
MatGetDiagonal(Mat A,Vec x);	$x = diag(A)$
MatTranspose(Mat A,Mat* B);	$B = A^T$
MatZeroEntries(Mat A);	$A = 0$
MatShift(Scalar *a,Mat Y);	$Y = Y + aI$

Вывести на экран матрицу можно командой

MatView(Mat mat,PETSC_VIEWER_STDOUT_WORLD);

Можно использовать и другие средства просмотра. Например, по умолчанию можно нарисовать ненулевую структуру матрицы в окне *X* командой

MatView(Mat mat,PETSC_VIEWER_DRAW_WORLD);

или использовать процедуру

MatView(Mat mat,PetscViewer viewer);

где просмотрщик был получен с помощью процедуры

ViewerDrawOpenX().

10.4. НЕМАТРИЧНОЕ ПРЕДСТАВЛЕНИЕ МАТРИЦ

Некоторые специалисты предпочитают использовать нематричные методы представления матриц, которые не требуют явного хранения матрицы при решения системы ДУЧП. Чтобы поддержать эти методы в PETSc, можно использовать следующие команды, которые создают *Mat* структуру без всякой реальной генерации матрицы:

MatCreateShell(MPI_Comm comm,int m,int n,int M,int N,void *ctx,Mat *mat);

Здесь *M* и *N* являются глобальными размерами матрицы (число строк и столбцов), *m* и *n* есть локальные размеры, а *ctx* – указателем на данные, необходимые любой определенной пользователем операции.

В справочных страницах имеются дополнительные детали об этих параметрах. Большинство нематричных алгоритмов требуют только приложения линейного оператора к вектору. Для этого пользователь может написать процедуру

UserMult(Mat mat,Vec x,Vec y);

и затем связать результат с матрицей *Mat*, используя команду

MatShellSetOperation(Mat mat,MatOperation MATOP_MULT,
(void(*)()) int (*UserMult)(Mat,Vec,Vec));

Здесь *MATOP_MULT* есть имя операции для матрично-векторного умножения. Внутри каждой определенной пользователем процедуры (такой, как UserMult()) пользователю нужно вызвать процедуру MatShellGetContext(), чтобы получить определенный пользователем контекст *ctx*, который был установлен процедурой MatCreateShell().

Процедуру MatShellSetOperation() можно использовать, чтобы установить любую другую матричную операцию. Файл *${PETSC_DIR}/include/petscmat.h* содержит полный список матричных операций, которые имеют форму MATOP_<OPERATION>, где <OPERATION> есть имя (все буквы заглавные) интерфейсной процедуры пользователя (например, MatMult()→ MATOP_MULT).

Все созданные пользователем функции имеют ту же последовательность обращения, что и обычные матричные интерфейсные процедуры, поскольку к пользовательским функциям предполагается обращаться через интерфейс, т. е.

MatMult(Mat,Vec,Vec) → UserMult(Mat,Vec,Vec).

Для MatShellSetOperation() последний аргумент должен быть типа void*, поскольку конечный аргумент (в зависимости от MatOperation) может быть множеством различных функций.

10.5. ДРУГИЕ МАТРИЧНЫЕ ОПЕРАЦИИ

Для многих итерационных вычислений (например, для решения нелинейных уравнений), чтобы повысить эффективность, важно повторно использовать ненулевую структуру матрицы, а не определять ее каждый раз, когда создается матрица. Чтобы оставить данную матрицу и повторно инициализировать, можно использовать процедуру

MatZeroEntries(Mat A);

Эта процедура будет обнулять матричные элементы в структуре данных, но сохранять все данные, которые указывают, где расположены ненулевые элементы. Следовательно, ассемблирование новой матрицы будет менее затратно, поскольку никакого распределения памяти или копирования не нужно. Конечно, можно и явно установить избранные матричные элементы в нуль, обратившись к процедуре MatSetValues().

В методах решения эллиптических уравнений в частных производных может оказаться слишком громоздким иметь дело с граничными условиями Дирихле. В частности, хотелось бы собрать матрицу без учета граничных условий, и затем в конце применить граничные условия Дирихле. В численном анализе обычно этот процесс представляют как перемещение граничных условий на правую сторону и затем решение линейной системы меньшей размерности для внутренних неизвестных. К сожалению, реализация этих требований приводит к извлечению большой субматрицы из исходной матрицы и созданию соответствующей ей структуры данных. Процесс может быть затратен как в отношении времени, так и в отношении памяти. Простой путь преодолеть трудность состоит в замещении строк матрицы, связанных с известными граничными условиями, строками идентичной матрицы (или с некоторым масштабированием ее). Для этого можно вызвать процедуру

MatZeroRows(Mat A,IS rows,Scalar *diag_value);

Для разреженных матриц это приводит к удалению структуры данных для определенных строк матрицы. Если указатель *diag_value* находится в состоянии PETSC_NULL, удаляются даже диагональные элементы. Матричная процедура

MatConvert(Mat mat,MatType newtype,Mat *M);

конвертирует матрицу *mat* в новую матрицу *M*, которая имеет тот же или другой формат.

Установка *newtype* в MATSAME для копирования матрицы сохраняет тот же формат матрицы. Другие доступные типы матриц описаны в файле *${PETSC_DIR}/include/petscmat.h.* Стандартными типами являются MATSEQDENSE, MATSEQAIJ, MATMPIAIJ, MATMPIROWBS, MATSEQBDIAG, MATMPIBDIAG, MATSEQBAIJ и MATMPIBAIJ. В определенных приложениях может оказаться необходимым для прикладных программ прямо обращаться к элементам матрицы. Это можно сделать по команде

MatGetRow(Mat A,int row, int *ncols,int **cols,Scalar **vals);

Аргумент *ncols* возвращает количество ненулевых элементов в строке, а *cols* и *vals* возвращают индексы столбцов и значения в строке. Если нужно указать только индексы столбца (а не соответствующие элементы), можно использовать PETSC_NULL для аргументов *vals* и *cols*. Пользователь может только исследовать значения, извлеченные при помощи MatGetRow(), значения не могут быть изменены. Чтоб изменить матричные элементы, необходимо использовать MatSetValues().

Чтобы освободить пространство, которое было занято при выполнении MatGetRow(), когда пользователь закончил использование строки, он обязан вызвать

MatRestoreRow(Mat A,int row,int *ncols,int **cols,Scalar **vals);

Глава 11. SLES: РЕШАТЕЛИ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЙ

11.1. ИСПОЛЬЗОВАНИЕ SLES

Пакет SLES является сердцем PETSc, поскольку он обеспечивает унифицированный и эффективный доступ ко всем средствам для решения систем линейных алгебраических уравнений (СЛАУ), включая параллельные и последовательные, прямые и итерационные методы. SLES предназначен для решения несингулярных систем формы

$$Ax = b, \qquad (11.1)$$

где A обозначает матричное представление линейного оператора; b – вектор правых частей; x – вектор решения.

SLES использует одинаковую последовательность обращений для прямых и итерационных методов решения линейных систем. Кроме того, во время решения могут быть выбраны частные способы решения и связанные с ними опции. Комбинация метода подпространств

Крылова (KSP – Krylov subspace method) и переобусловливателя (PC – preconditioner) находится в центре самых современных программ для решения линейных систем итерационными методами. SLES создает упрощенный интерфейс к низкоуровневым модулям KSP и PC внутри пакета PETSc. Компонент KSP предоставляет многие известные итерационные методы на основе подпространств Крылова. Модуль PC включает разные переобусловливатели. Хотя KSP и PC могут использоваться непосредственно, пользователям лучше применять интерфейс SLES.

Чтобы решить линейную систему с помощью SLES, сначала нужно создать контекст решателя командой

SLESCreate(MPI_Comm comm,SLES *sles);

Здесь *comm* есть коммуникатор MPI; *sles* – вновь сформированный контекст решателя.

Перед действительным решением линейной системы с помощью SLES пользователь обязан вызвать следующую процедуру, чтобы задать матрицы, связанные с линейной системой:

SLESSetOperators(SLES sles,Mat Amat,Mat Pmat,MatStructure flag);

Аргумент *Amat*, представляющий матрицу, которая определяет линейную систему, определяет место для матриц любого вида. В частности, SLES поддерживает нематричный метод представления матриц. Как правило, матрица переобусловливания *Pmat* (т. е. матрица, из которой должен быть сконструирован переобусловливатель), является той же, что и матрица *Amat*, которая определяет линейную систему, однако иногда эти матрицы отличаются. Аргумент *flag* может быть использован, чтобы исключить ненужную работу, когда многократно решаются линейные системы того же размера и тем же методом переобусловливания. Когда решается только одна система, этот флаг игнорируется. Пользователь может установить *flag* следующим образом.

- SAME_NONZERO_PATTERN – матрица переобусловливания имеет ту же самую ненулевую структуру на протяжении последовательных линейных решений.
- DIFFERENT_NONZERO_PATTERN – матрица переобусловливания не имеет той же самой ненулевой структуры на протяжении последовательных линейных решений.
- SAME_PRECONDITIONER – матрица переобусловливания идентична матрице предыдущего линейного решения.

• Если имеется сомнение относительно структуры матрицы, нужно использовать флаг DIFFERENT_NONZERO_PATTERN.

Полезна процедура

SLESSetFromOptions(SLES sles);

Эта процедура имеет опции -h и -help так же, как и любые KSP и PC опции, обсуждаемые ниже. Чтобы решить линейную систему, в большинстве случаев выполняют команду

SLESSolve(SLES sles,Vec b,Vec x,int *its);

где *b* и *x* соответственно обозначают вектор свободных членов и вектор решения. Параметр *its* содержит либо номер итерации, на которой сходимость была успешно достигнута, или номер итерации, на которой была обнаружена расходимость или прерывание. Заметим, что многократные решения могут быть получены с тем же SLES контекстом. Когда контекст больше не нужен, он удаляется командой

SLESDestroy(SLES sles);

Приведенная процедура хороша для общего использования пакета SLES. Однако требуется дополнительный шаг для пользователей, которые хотят настроить определенные переобусловливатели или просмотреть определенные характеристики, используя профилирующие возможности PETSc. В nfrjv случае пользователь может вызвать

SLESSetUp(SLES sles,Vec b,Vec x);

перед вызовом процедуры SLESSolve(), чтобы выполнить любые установки, требуемые для линейных решателей. Явный вызов этой процедуры устанавливает отдельный мониторинг за любыми вычислениями на протяжении установленной фазы, такой, как неполная факторизации для ILU переобусловливателя.

Решатель, который запускается в SLES по умолчанию, GMRES (**G**eneralized **M**inimal **R**esidual) – обобщенный метод минимальной невязки. Он переобусловлен для последовательного случая с ILU(0) и для параллельного случая с блочным методом Якоби (с одним блоком на процесс, каждый из которых работает с ILU(0)). Также доступны и многие другие решатели и опции. Чтобы позволить прикладным программистам установить любую опцию из переобусловливателей или метода подпространств Крылова прямо внутри программы, предлагаются процедуры, которые извлекают PC или KSP контекст:

SLESGetPC(SLES sles,PC *pc);
SLESGetKSP(SLES sles,KSP *ksp);

Пользователи могут прямо вызвать любую из процедур PC или KSP, чтобы модифицировать соответствующую (по умолчанию) опцию. Чтобы решить линейную систему прямым методом (в настоящее время поддерживается только для последовательных матриц), можно использовать опции -pc_type lu, -ksp_type preonly.

По умолчанию, если используется прямой решатель, при факторизации не делается замещения (in-place). Это должно защитить пользователя от повреждения матриц после окончания решения. Процедура PCLU SetUseInPlace(), обсуждаемая ниже, позволяет сделать факторизацию с замещением.

11.2. РЕШЕНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ЛИНЕЙНЫХ СИСТЕМ

Для многократного решения линейных систем того же размера и тем же методом можно использовать несколько опций. Чтобы решать последовательные системы, имея ту же матрицу переобусловливателя (т. е. ту же структуру данных с точно такой же матрицей элементов, но с различными свободными членами), пользователю следует просто вызвать процедуру SLESSolve() несколько раз. Переобусловливатель устанавливает операции (например, факторизацию для ILU), которые будут выполняться только в течение первого обращения к SLESSolve(), такие операции не будут выполняться для последующих решений. Чтобы решать последовательные линейные системы, которые имеют различные матрицы переобусловливателей (т. е. матричные элементы и/или структуры данных матриц изменены), пользователь обязан вызвать процедуры SLESSetOperators() и SLESSolve() для каждого решения.

11.3. МЕТОДЫ КРЫЛОВА

Для методов подпространств Крылова имеется ряд опций, многие из них обсуждаются ниже. Чтобы указать метод Крылова, который будет использоваться, нужно вызвать команду

KSPSetType(KSP ksp,KSPType method);

Тип для *method* может быть: KSPRICHARDSON, KSPCG, KSPGMRES, KSPTCQMR, KSPCHEBYCHEV, KSPBCGS, KSPCGS, KSPTFQMR, KSPCR, KSPLSQR, KSPBICG или KSPPREONLY. Метод KSP также можно установить командами базы опций -ksp_type, сопровождаемыми одной из опций: richardson, chebychev, cg, gmres, tcqmr, bcgs, cgs, tfqmr, cr, lsqr, bicg или preonly. Имеются специфические опции для методов Ричардсона, Чебышева и GMRES:

KSPRichardsonSetScale(KSP ksp,double damping_factor);
KSPChebychevSetEigenvalues(KSP ksp,double emax,double emin);
KSPGMRESSetRestart(KSP ksp,int max_steps);

Значениями параметров по умолчанию являются: *damping_factor* = 1.0, *emax* = 0.01, *emin*= 100.0 и *max_steps* = 30. Дампинг фактор для рестарта GMRES и метода Ричардсона устанавливается также опциями -ksp_gmres_restart<n> и -ksp_richardson_scale<factor>.

По умолчанию для ортогонализации матрицы Гезенберга в GMRES используется итерационное уточнение по методу Gram-Schmidt'a. Это можно установить опцией командной строки -ksp_-gmres_irorthog или с помощью

KSPGMRESSetOrthogonalization(KSP ksp,
KSPGMRESModifiedGramSchmidtOrthogonalization);

Более быстрые подходы используют немодифицированный метод Gram-Schmidt'a, который можно установить командой

KSPGMRESSetOrthogonalization(KSP ksp,
KSPGMRESUnmodifiedGramSchmidtOrthogonalization);

или командой базы опций -ksp_gmres_unmodifiedgramschmidt. Заметим, что этот алгоритм численно неустойчив, но имеет несколько лучшие скоростные характеристики. Можно также использовать модифицированный метод Gram-Schmidt'a, устанавливая процедуру ортогонализации KSPGMRESModifiedGramSchmidtOrthogonalization или используя -ksp_gmres_modifiedgramschmidt опцию командной строки.

Для метода сопряженных градиентов с комплексными числами имеются два немного отличающихся алгоритма в зависимости от того, является ли матрица Hermitian симметрической или истинно симметрической (по умолчанию она считается Hermitian симметрической). Чтобы указать вид симметрии, используется команда

KSPCGSetType(KSP ksp,KSPCGType KSP_CG_SYMMETRIC);

Алгоритм LSQR не использует переобусловливатель. Любой переобусловливатель, установленный для работы с объектами KSP, игнорируется, если выбран LSQR. По умолчанию KSP предполагает нулевое начальное значение, получаемое обнулением начального значения для заданного вектора решения. Это обнуление выполняется обращением к процедуре SLESSolve() (или KSPSolve()). Чтобы использовать ненулевое начальное приближение, нужно вызвать

KSPSetInitialGuessNonzero(KSP ksp);

11.3.1. Переобусловливание внутри KSP

Порядок сходимости проективных методов Крылова для линейных систем сильно зависит от их спектра. Для изменения спектра и, следовательно, для ускорения сходимости итерационных методов, как правило, используется переобусловливание, которое может быть применено к системе (11.1) путем преобразования

$$\left(M_L^{-1} A M_R^{-1}\right)\left(M_R x\right) = M_L^{-1} b\,, \qquad (11.2)$$

где M_L и M_R указывают матрицу переобусловливания (или матрицу, из которой должен быть сконструирован переобусловливатель).

Если $M_L = I$ в 11.2, то правый переобусловливающий результат и невязка

$$r \equiv b - Ax = b - AM_R^{-1} M_R x$$

сохраняются. Напротив, невязка изменяется для left($M_L = I$) и симметричного переобусловливливания согласно выражению

$$r_L \equiv M_L^{-1} b - M_L^{-1} Ax = M_L^{-1} r.$$

По умолчанию все реализации KSP используют левое переобусловливание. Правое переобусловливание может быть активировано для некоторых методов командой базы опций -ksp_right_pc или вызовом процедуры

KSPSetPreconditionerSide(KSP ksp,PCSide PC_RIGHT);

В табл. 11.1 указаны все методы KSP, используемые по умолчанию, и по умолчанию они с левым переобусловливанием. Попытка использовать правое переобусловливание для методов, которые в настоящее время не поддерживаются, приводит к ошибке вида

KSPSetUp_Richardson:No right preconditioning for KSPRICHARDSON.

Переобусловленная невязка используется по умолчанию для тестирования сходимости всех левопереобусловленных методов, кроме метода сопряженных градиентов и методов Ричардсона, и Чебышева. Для этих трех случаев истинная невязка использована по умолчанию, но вместо этого может быть использована переобусловленная невязка с командой базы опций -ksp_preres или вызовом процедуры

KSPSetUsePreconditionedResidual(KSP ksp);

Таблица 11.1.

Методы KSP

Метод	Тип KSP	Имя в базе данных	Тип сходимости по умолчанию
Richardson	KSPRICHARDSON	richardson	true
Chebychev	KSPCHEBYCHEV	chebychev	true
Conjugate Gradient	KSPCG	cg	true
BiConjugate Gradient	KSPBICG	bicg	true
Generalized Minimal Residual	KSPGMRES	gmres	precond
BiCGSTAB	KSPBCGS	bcgs	precond
Conjugate Gradient Squared	KSPCGS	cgs	precond
Transpose-Free-Quasi-Minimal Residual (1)	KSPTFQRM	tfqrm	precond
Transpose-Free-Quasi-Minimal Residual (2)	KSPTCQRM	tcqrm	precond
Conjugate Residual	KSPCR	cr	precond
Least Squares Method	KSPLSQR	lsqr	precond
Shell for no KSP Method	KSP		precond

true обозначает реальную норму невязки;
precond –переобусловленную норму невязки.

11.3.2. Тесты сходимости

Используемый тест KSPDefaultConverged() основан на норме l_2 невязки. Сходимость (или расходимость) характеризуется тремя величинами: относительным уменьшением нормы невязки *rtol*, абсолютной величиной нормы невязки *atol* и относительным повышением в невязке *dtol*. Сходимость обнаруживается на итерации k, если

$$\|r_k\|_2 < \max\left(rtol * \|r_0\|_2, atol\right),$$

где $r_k = b - Ax_k$. Расходимость определена, если

$$\|r_k\|_2 > dtol * \|r_0\|_2.$$

Эти параметры, так же,как и максимальное количество допустимых итераций, можно установить процедурой

KSPSetTolerances(KSP ksp,double rtol,double atol,double dtol,int maxits);

Пользователь может оставить значение, выбранное по умолчанию для любого из этих параметров описав PETSC_DEFAULT, как соответствующий допуск. По умолчанию эти величины имеют следующие значения: *rtol* = 10^{-5}, *atol* = 10^{-50}, *dtol* = 10^{5} и *maxits* = 10^{5}. Их также можно изменить командами:

-ksp_rtol<rtol>
-ksp_atol<atol>
-ksp_divtol <dtol>
-ksp_max_it <its>

В дополнение к интерфейсу для простых тестов сходимости KSP предоставляет пользователям возможность настройки процедур тестирования сходимости. Пользователь может указать настроечную процедуру командой

```
KSPSetConvergenceTest(KSP ksp,int (*test)(KSP ksp,int it,double rnorm,
                      PConvergedReason *reason,void *ctx), void *ctx);
```

Конечный аргумент процедуры *ctx* является опцией контекста для частных данных определенной пользователем процедуры сходимости *test*. Другими аргументами процедуры *test* являются количество итераций *it* и норма l_2 невязки *rnorm*. Процедура *test* должна установить положительное значение при наличии сходимости, 0 – если нет сходимости, и отрицательное значение в случае расходимости. Список возможных KSPConvergedReason приводится в *include/petscksp.h.*

11.3.3. Мониторинг сходимости

По умолчанию решатели Крылова работают без вывода на экран информации об итерациях. Пользователь может указать с помощью опции -ksp_monitor внутри базы опций необходимость вывода на экран нормы невязки. Чтобы изобразить их в графическом окне (работая под X Windows), следует использовать опцию -ksp_xmonitor[x,y,w,h], где должны быть описаны все варианты, или ни одного. Пользователи также могут использовать собственные процедуры для выполнения мониторинга, используя команду

```
KSPSetMonitor(KSP ksp,int (*mon)(KSP ksp,int it,double
                rnorm,void *ctx), void *ctx,int (*mondestroy)(void *));
```

Последний аргумент процедуры *ctx* есть опция контекста для частных данных для определенной пользователем процедуры *mon*. Другие аргументы этой процедуры – это количество итераций (*it*) и норма

l_2 невязки *rnorm*. Полезной процедурой среди определенных пользователем мониторов является

```
PetscObjectGetComm ((PetscObject)ksp,MPI_Comm*comm);
```

которая возвращает в *comm* значение коммуникатора MPI для контекста KSP. Вместе с PETSc поставляется несколько процедур мониторинга, включая

```
KSPDefaultMonitor(KSP,int,double, void *);
KSPSingularValueMonitor(KSP,int,double, void *);
KSPTrueMonitor(KSP,int,double, void *);
```

Установленный по умолчанию монитор просто выводит оценку нормы l_2 невязки на каждой итерации.

Процедура KSPSingularValueMonitor пригодна только для использования с методом сопряженных градиентов или GMRES, поскольку она выводит оценки экстремальных сингулярных значений переобусловленного оператора на каждой итерации. Так как процедура KSPTrueMonitor выводит истинную невязку на каждой итерации путем действительного вычисления, используя формулу $r = b – Ax$, эта процедура медленная, и ее следует использовать для тестирования или изучения сходимости, но не для получения малого времени вычислений. Опции командной строки

```
-ksp_monitor
-ksp_singmonitor
-ksp_truemonitor
```

позволяют обратиться к этим мониторам.

Чтобы использовать установленный по умолчанию графический монитор, вызываем команды

```
PetscDrawLG lg;
KSPLGMonitorCreate(char *display,char *title,int x,
                   int y,intw,int h, PetscDrawLG *lg);
KSPSetMonitor(KSP ksp,KSPLGMonitor,lg,0);
```

Если монитор больше не нужен, он удаляется командой

```
KSPLGMonitorDestroy(PetscDrawLG lg);
```

Пользователь может изменить параметры графики с помощью процедур DrawLG*() и DrawAxis*() или с помощью команд базы опций -ksp_xmonitor[x,y,w,h], где *x*, *y*, *w*, *h* по умолчанию являются ячейкой и размером окна. Жестко установленные процедуры монито-

ринга для KSP отменяются во время исполнения с помощью -ksp_ cancelmonitors.

Поскольку сходимость метода Крылова такая, что норма невязки мала (например, 10^{-10}), много конечных цифр, выведенных опцией -ksp_monitor, теряют смысл. Более того, они отличаются на разных машинах из-за правил округления. Это затрудняет тестирование для машин. Опция -ksp_smonitor заставлят PETSc выводить меньше цифр нормы невязки, когда она становится малой. Поэтому на большинстве машин будут выводится те же числа, что облегчает тестирование.

11.3.4. Спектр операторов

Сходимость метода Крылова прямо зависит от спектра собственных значений переобусловленного опратора, поэтому PETSc имеет специфические процедуры для аппроксимации собственных значений с помощью итерационных методов Arnoldi или Lanczos'a. Сначала, перед линейным решением, обязательно вызывают

```
KSPSetComputeEigenvalues (KSP ksp);
```

Затем после решения SLES вызывают процедуру

```
KSPComputeEigenvalues(KSP ksp, int n,double *realpart,
                      double *complexpart,int *neig);
```

Здесь *n* – размер двух массивов, в которых размещаются собственные значения. Аргумент *neig* есть количество вычисленных собственных значений. Эта числовая зависимость определяется размером пространства Крылова, сгенерированного во время решения линейной системы; для GMRES она никогда не превышает параметр рестарта.

Дополнительная процедура

```
KSPComputeEigenvaluesExplicitly(KSP ksp, int n,double *realpart,
                                double *complexpart);
```

полезна для маленьких задач (до пары сотен строк и столбцов). Она явно вычисляет полное представление переобусловленного оператора и вызывает LAPACK для вычисления собственных значений. Процедуры DrawSP*() полезны для представления рассылаемых частей собственных значений.

Собственные значения также можно вычислить и вывести на экран с помощью команд базы опций:

```
-ksp_plot_eigenvalues
-ksp_plot_ eigenvalues_explicitly
```

или для кода ASCII – через опции:

-ksp_compute _eigenvalues
-sp_compute_eigenvalues_explicitly

11.3.5. Другие опции KSP

Чтобы получить вектор решения и правую часть из контекста KSP, используют процедуры

KSPGetSolution(KSP ksp,Vec *x);
KSPGetRhs(KSP ksp,Vec *rhs);

На протяжении итерационного процесса решение еще может быть не вычислено или храниться в разных местах. Чтобы получить доступ к приближенному решению во время итерационного процесса, используют команду

KSPBuildSolution(KSP ksp,Vec w,Vec *v);

где решение возвращается в *v*. Пользователь может разместить вектор в переменной *w*, как в месте хранения вектора. Однако если *w* имеет значение PETSC_NULL, используется пространство, распределенное PETSc в KSP контексте. Для некоторых методов KSP (например, для GMRES) построение решения затратно, но для многих других методов не требуется даже копирования вектора. Доступ к невязкам производится в подобном случае командой

KSPBuildResidual(KSP ksp,Vec t,Vec w,Vec *v);

Для GMRES и некоторых других методов это затратная операция.

11.4. ПЕРЕОБУСЛОВЛИВАТЕЛИ

Как указывалось в параграфе 11.3.1, метод подпространств Крылова обычно используется в сочетании с переобусловливателем. Чтобы использовать частный метод переобусловливливания, пользователь может выбрать его из базы опций, используя вход вида -pc_type <methodname> или установить метод командой

PCSetType(PC pc,PCType method);

В табл. 11.2 представлены основные методы переобусловливания, поддерживаемые в PETSc.

Переобусловливатель PCSHELL использует прикладной специфический переобусловливатель. Прямой переобусловливатель есть фактически прямой решатель линейной системы, который использует LU-факторизацию.

Таблица 11.2 .

Переобусловливатели PETSc

Метод	Тип переобуслов-ливателя	Имя в базе опций
Jacobi	PCJACOBI	jacobi
Block Jacobi	PCBJACOBI	bjacobi
SOR (and SSOR)	PCSOR	sor
SOR with Eisenstat trick	PCEISENSTAT	eisenstat
Incomplete Cholesky	PCICC	icc
Incomplete LU	PCILU	ilu
Additive Schwarz	PCASM	asm
Linear Solver	PCSLES	sles
Combination of preconditioners	PCCOMPOSITE	composite
LU	PCLU	lu
Cholesky	PCCholesky	cholesky
No preconditioning	PCNONE	none
Shell for user-defined PC	PCShell	shell

PCLU включен как переобусловливатель, чтобы PETSc имел содержательный интерфейс для прямых и итерационных линейных решателей. Каждый переобусловливатель имеет ряд связанных с ним опций, которые могут быть установлены процедурами и опциями команд базы данных. Все имена таких процедур и команд имеют форму PC<TYPE>Option и -pc_<type>_option[value].

Полный список имеется в справочных страницах.

11.4.1. ILU и ICC переобусловливатели

Некоторыми опциями для переобусловливателей ILU являются

PCILUSetLevels(PC pc,int levels);
PCILCCSetLevels(PC pc,int levels);
PCILUSetReuseOrdering(PC pc,PetscTruth flag);
PCILUSetUseDropTolerance(PC pc,double dt,int dtcount);
PCILUDTSetReuseFill(PC pc,PetscTruth flag);
PCILUSetUseInPlace(PC pc);
PCILUSetAllowDiagonalFill(PC pc);

Когда многократно решаются линейные системы с тем же контекстом SLES, можно повторно использовать информацию первого решения. В частности, процедура PCILUSetReuseOrdering() вызывает упорядочивание (например, установкой опции -pc_ilu_ordering_ typeorder), вычисленное в первой факторизации, которое должно быть использовано повторно в поздних факторизациях.

Процедура PCILUSetUseInPlace() часто используется с методами PCASM или PCBJACOBI при нулевом заполнении, поскольку она повторно использует матричное пространство для сохранения полной факторизации, что экономит память и время. Заметим, что факторизация с повторным использованием пространства не соответствует никакому упорядочиванию, кроме натурального, и не может быть использована с понижением допуска факторизации. Для этого используются опции

-pc_ilu_levels <levels>
-pc_ilu_reuse_ordering
-pc_ilu_use_drop_tolerance <dt>,<dtcount>
-pc_ilu_reuse_fill
-pc_ilu_in_place
-pc_ilu_nonzeros_along_diagonal
-pc_ilu_diagonal_fill

Для уменьшения значительных накладных расходов при динамическом распределении памяти такая настройка может существенно улучшить характеристики.

PETSc поддерживает неполные переобусловливатели для нескольких матричных типов в последовательном случае. Для параллельного случая предоставляется интерфейс для ILU и ICC переобусловливателей из BlockSolve95. Подробности применения BlockSolve95 изложены в файле *docs/installation/index.htm*. PETSc разрешает использовать переобусловливатели внутри BlockSolve95, используя матричный формат MATMPIROWBS из BlockSolve95 и привлекая методы PCILU или PCICC внутри линейных решателей.

Поскольку PETSc автоматически выполняет сборку матриц, установку переобусловливателей, профилирование и другие, пользователям, которые обращаются к BlockSolve95 через интерфейс PETSc, не нужно заботиться о многих деталях, описанных в руководстве пользователя по BlockSolve95. Матрица, совместимая с BlockSolve95, создается при помощи MatCreate() с опцией -mat_mpirowbs, или прямым вызовом процедуры

MatCreateMPIRowbs(MPI_Comm comm,int m,int M,int nz,int *nnz,Mat *A).

Здесь A – вновь созданная матрица, а аргументы m и M указывают соответственно количество локальных и глобальных строк. Локальные и глобальные параметры могут быть установлены с помощью введения параметра PETSC_DECIDE, так что их определять будет PETSc. Аргументы *nz* и *nnz* могут быть использованы для предраспределения пространства памяти для повышения эффективности сборки матрицы. Аргументы *nz* = 0 и *nnz* = PETSC_NULL для PETSc устанавливаются, чтобы управлять всем размещением матричной памяти.

Если матрица симметрическая, для повышения эффективности можно вызвать следующую функцию

MatSetOption(Mat mat,MAT_SYMMETRIC);

но в таком случае нельзя использовать ILU переобусловливатели, а только ICC.

Внутри PETSc, если необходимо, нулевые элементы размещаются в матрицах формата MATMPIROWBS, так что несиметрические матрицы рассматриваются как симметрические. Этот формат требуется для параллельных коммуникационных процедур внутри BlockSolve95. BlockSolve95 при операциях с переобусловливающей матрицей A внутри работает с масштабируемой матрицей $\widehat{A} = PD^{-1/2}AD^{-1/2}$, элементы которой переставлены. Здесь D – диагональ матрицы A, а P – переставленная матрица, определяемая графом раскраски для эффективного параллельного вычисления.

Поэтому, когда решается линейная система $Ax = B$ с использованием ILU/ICC переобусловливания и матричного формата MATMPIROWBS, для матрицы линейной системы и переобусловливающей матрицы на самом деле решается масштабируемая и переставленная система $\widehat{A}\widehat{x} = \widehat{b}$, где $\widehat{x} = PD^{1/2}x$ и $\widehat{b} = PD^{-1/2}b$.

PETSc выполняет внутреннее масштабирование и перестановку x и b, поэтому пользователь не имеет дела с этими преобразованиями, а всегда работает с оригинальной линейной системой. В таком случае мониторинг по умолчанию выполняется для масштабируемой нормы невязки. Чтобы вывести масштабируемую и немасштабируемую нормы невязки, нужна опция -ksp_truemonitor.

11.4.2. SOR and SSOR переобусловливатели

PETSc не обеспечивает параллельного решения методом SOR (последовательной повторной релаксации), он может быть использован

только на последовательных матрицах или на субблоках переобусловливателя, когда используется блочный метод Якоби или ASM переобусловливание.

Опциями для SOR переобусловливания являются

PCSORSetOmega(PC pc,double omega);
PCSORSetIterations(PC pc,int its);
PCSORSetSymmetric(PC pc,MatSORType type);

Первая из этих команд устанавливает параметр релаксации для последовательной верхней (или нижней) релаксации. Вторая команда устанавливает количество внутренних итераций SOR, задаваемых *its* для использования между шагами метода Крылова. Третья команда устанавливает тип развертки SOR:

SOR_FORWARD_SWEEP
SOR_BACKWARD_SWEEP
SOR_SYMMETRIC_SWEEP

По умолчанию используется тип SOR_FORWARD_SWEEP. Установка типа SOR_SYMMETRIC_SWEEP приводит к методу SSOR. Каждый процесс также может независимо и локально выполнять описанный вариант SOR с типами:

SOR_LOCAL_FORWARD_SWEEP,
SOR_LOCAL_BACKWARD_SWEEP,
SOR_LOCAL_SYMMETRIC_SWEEP.

Для этих вариантов также есть опции

-pc_sor_omega<omega> **-pc_sor_its<its>**
-pc_sor_backward **-pc_sor_symmetric,**
-pc_sor_local_forward **-pc_sor_local_backward,**
-pc_sor_local_symmetric

11.4.3. LU факторизация

LU переобусловливатели имеют несколько опций. Первая задается командой

PCLUSetUseInPlace(PC pc);

и вызывает факторизацию с замещением, т. е. с разрушением исходной матрицы. Опцией этой команды является -pc_lu_in_place. Другая опция для прямого переобусловливателя, выбирающая упорядочение уравнений, работает по команде

-pc_lu_ordering_type <ordering>

Возможные упорядочения таковы:

- MATORDERING_NATURAL – Natural,
- MATORDERING_ND – Nested Dissection,
- MATORDERING_1WD – One-way Dissection,
- MATORDERING_RCM – Reverse Cuthill-McKee,
- MATORDERING_QMD – Quotient Minimum Degree.

Эти упорядочивания могут быть также установлены опциями:

-pc_lu_ordering_type natural
-pc_lu_ordering_type nd
-pc_lu_ordering_type 1wd
-pc_lu_ordering_type rcm
-pc_lu_ordering_type qmd

Разреженная (sparse) LU факторизация, предлагаемая в PETSc, не выполняет выбор главного элемента для обеспечения стабильности вычислений, поскольку они спроектированы, чтобы сохранить ненулевую структуру. Поэтому время от времени LU факторизация будет переставать работать с нулевым главным элементом, когда в действительности матрица несингулярная.

Часто опция -pc_lu_nonzeros_along_diagonal<tol> помогает исключить нулевой главный элемент благодаря предобработке столбцов, чтобы удалить небольшие значения из диагонали. Здесь *tol* есть допуск опции, позволяющий решить, является ли значение ненулевым. По умолчанию допуск равен 10^{-10}.

11.4.4. Блочный переобусловливатель Якоби и аддитивный переобусловливатель Шварца с перекрытием

Блочный метод Якоби и аддитивный метод Шварца с перекрытием поддерживаются параллельно. В настоящее время представлена только последовательная версия метода Гаусса–Зейделя. По умолчанию PETSc реализует эти методы, используя ILU(0) на каждом индивидуальном блоке. Это означает, что по умолчанию на каждом субблоке используется решатель PCType = PCILU, KSPType = KSPPREONLY. Пользователь может установить альтернативные линейные решатели опциями -sub_ksp_type и -sub_pc_type. В действительности все KSP и PC опции могут быть применены к подзадачам путем размещения преффикса -sub_ в начале имени опции. Эти команды базы опций устанавливают частные опции для всех блоков внутри глобальной задачи.

Дополнительно процедуры

PCBJacobiGetSubSLES(PC pc,int *n_local,int *first_local,SLES **subsles);
PCASMGetSubSLES(PC pc,int *n_local,int *first_local,SLES **subsles);

извлекают SLES контекст для каждого локального блока. Аргумент *n_local* задает количество блоков в вызывающем процессе, а *first_local* указывает глобальный номер первого блока в процессе. Блоки нумеруются процессами последовательно от нуля до *gb* – 1, где *gb* есть количество глобальных блоков. Массив контекстов SLES для локальных блоков задается аргументом *subsles*. Этот механизм позволяет пользователю установить разлные решатели для различных блоков. Чтобы установить соответствующие структуры данных, пользователь обязан вызвать SLESSetUp() перед вызовом PCBJacobiGetSubSLES() или PCASMGetSubSLES(). Подробности рассматриваются в *${PETSC_DIR}/src/sles/examples/tutorials/ex7.c*.

Переобусловливатели: блочный Якоби, блочный Гаусса-Зейделя и аддитивный Шварца, позволяют пользователю установить количество блоков, на которые делится задача. Опции для установления этих значений таковы: -pc_bjacobi_blocksn и -pc_bgs_blocksn, а внутри программ работают соответствующие процедуры:

PCBJacobiSetTotalBlocks(PC pc,int blocks,int *size);
PCASMSetTotalSubdomains(PC pc,int n,IS *is);
PCASMSetType(PC pc,PCASMType type);

Аргумент опции *size* есть массив, указывающий размер каждого блока. В настоящее время для определенных форматов матриц поддерживается только единственный блок на процесс. Однако форматы MATMPIAIJ и MATMPIBAIJ поддерживают использование общих блоков до тех пор, пока не станет блоков, разделяемых между процессами. Аргумент *is* содержит индексные ряды, которые определяют подобласти.

Тип PCASMType имеет одно из значений:

PC_ASM_BASIC,
PC_ASM_INTERPOLATE,
PC_ASM_RESTRICT,
PC_ASM_NONE

и может также быть установлен опцией -pc_asm_type [basic,interpolate, restrict,none]. Тип PC_ASM_BASIC соответствует стандартному аддитивному методу Шварца. Тип PC_ASM_RESTRICT установлен в PETSc по умолчанию, поскольку он сохраняет реальную коммуника-

цию и для многих задач дает дополнительную выгоду, поскольку требует меньше итераций для получения сходимости, чем стандартный метод Шварца.

Пользователь также может установить количество блоков и их размеры командами

PCBJacobiSetLocalBlocks(PC pc,int blocks,int *size);
PCASMSetLocalSubdomains(PC pc,int N,IS *is);

Для переобусловливателя ASM можно использовать следующую команду, чтобы установить перекрытие подобластей

PCASMSetOverlap(PC pc,int overlap);

Перекрытие по умолчанию равно 1, поэтому если дополнительное перекрытие не нужно (оно может быть установлено вызовами процедур PCASMSetTotalSubdomains() или PCASMSetLocal Subdomains()), то аргумент *overlap* должен быть установлен в 0. В частности, если явно не установлены подобласти в прикладном коде, то любое перекрытие было бы вычислено внутри PETSc и, используя перекрытие 0, результировалось бы в ASM варианте, который эквивалентен блочному переобусловливателю Якоби. Заметим, что можно определить начальные индексные ряды *is* с любым перекрытием с помощью процедур PCASMSetTotalSubdomains() или PCASMSetLocalSubdomains(). Процедура PCASMSet Overlap(), если нужно, позволяет PETSc просто расширить такое перекрытие.

11.4.5. Переобусловливатели на основе оболочки

Переобусловливатель на основе оболочки (shell) просто использует прикладную процедуру, чтобы реализовать нужную процедуру. Установить эту процедуру можно командой

PCShellSetApply(PC pc,int (*apply)(void *ctx,Vec,Vec),void *ctx);

Конечный аргумент *ctx* – указатель на прикладную структуру данных, необходимую для процедуры переобусловливателя. Три аргумента apply() являются контекстом, соответственно входным и выходным векторами. Для переобусловливателя, который требует некоторой «установки» перед использованием, можно создать «установочную» процедуру, которая вызывается всегда, если изменяется оператор (обычно через SLESSetOperators()):

PCShellSetSetUp(PC pc,int (*setup)(void *ctx));

Аргументом для процедуры является та же самая прикладная структура данных, переданная процедурой PCShellSetApply().

11.4.6. Объединение переобусловливателей

Во многих случаях использовать одиночный переобусловливатель лучше, чем комбинационный. Исключение составляют многосеточные или многоуровневые переобусловливатели (решатели), которые всегда являются некоторого вида комбинацией.

Пусть B_1 и B_2 представляют два переобусловливателя типа type1 и type2. Тогда переобусловливатель $B = B_1 + B_2$ может быть получен с помощью

```
PCSetType(pc,PCCOMPOSITE);
PCCompositeAddPC(pc,type1);
PCCompositeAddPC(pc,type2);
```

Этим способом можно добавить любое количество переобусловливателей. Такой путь объединения называется аддитивным, поскольку действия переобусловливателей складываются вместе. Он установлен по умолчанию. Альтернативой может быть установка с опцией

```
PCCompositeSetType(PC pc,PCCompositeType
                   PC_COMPOSITE_MULTIPLICATIVE);
```

В этой форме новая невязка обновляется после применения каждого переобусловливателя, и следующий переобусловливатель применяется к следующей невязке. Например, пусть имеются два переобусловливателя, тогда $y = Bx$ получается из

$$y = B_1 x, \qquad w_1 = x - Ay, \qquad y = y + B_2 w_1.$$

Приближенно это соответствует итерациям метода Гаусса–Зейделя, а аддитивный вариант – похож на метод Якоби.

Благодаря многим обстоятельствам мультикативная форма требует вдвое меньше итераций, чем аддитивная, но мультикативная форма делает необходимым применение A внутри переобусловливателя. В мультипликативной версии вычисление невязки внутри переобусловливателя можно выполнить двумя путями: используя исходную матрицу линейной системы или матрицу для построения переобусловливателей $B1$, $B2$ и т. д. По умолчанию используется «матрица переобусловливателя». Чтобы использовать истинную матрицу, нужно выбрать опцию

```
PCCompositeSetUseTrue(PC pc);
```

Доступ к индивидуальным переобусловливателям может быть осуществлен процедурой

PCCompositeGetPC(PC pc,int count,PC *subpc);

Например, чтобы установить сначала подмножество переобусловливателей для работы с ILU(1), нужно выполнить

```
PC subpc;
PCCompositeGetPC(pc,0,&subpc);
PCILUSetFill(subpc,1);
```

Для этого можно использовать базу опций. Например, опции:

```
-pc_typecomposite
-pc_composite_pcsjacobi,ilu
```

вызывают использование композитного переобусловливателя с двумя переобусловливателями: Jacobi и ILU.

Опция

-pc_composite_typemultiplicative

инициирует мультипликативную версию алгоритма, а опция

-pc_composite_typeadditive

вызывает аддитивную версию.

Использование истинного переобусловливателя получается с помощью опции

-pc_composite_true.

Можно установить опции для субпереобусловливателей с дополнительным преффиксом -sub_N_, где *N* есть номер субпереобусловливателя. Например, -sub_0_pc_ilu_fill0.

PETSc также позволяет переобусловливателю быть законченным линейным решателем. Это достигается с помощью типа PCSLES:

```
PCSetType(PC pc,PCSLES PCSLES);
PCSLESGetSLES(pc,&sles);
        /* set any SLES/KSP/PC options */
```

По умолчанию внутренний SLES переобусловливатель использует самую внешнюю (outter) «матрицу переобусловливателя» как матрицу, решаемую в линейной системе. Чтобы использовать истинную матрицу, используется опция

PCSLESSetUseTrue(PC pc);

или из командной строки -pc_sles_true.

ПРИЛОЖЕНИЕ

НЕКОТОРЫЕ ТЕРМИНЫ РАЗДЕЛА 3

Additive Schwarz – аддитивный метод Шварца
ASM (Additive Schwarz) – аддитивный метод Шварца для переобусловливателя
BICG (BiConjugate Gradient) – метод бисопряженных градиентов
BiCGStab – устойчивый метод бисопряженных градиентов
BLAS (Basic Linear Algebra Subprograms) – библиотека для работы с векторами и матрицами
Block Jacobi – блочный метод Якоби
CG (Conjugate Gradient) – метод сопряженных градиентов
Chebychev – метод Чебышева
Cholesky – метод Холесского
EISENSTAT (SOR with Eisenstat trick) – метод SOR с приемом Айзенштадта
GMRES (Generalized Minimal Residual) – обобщенный метод минимальной невязки
ICC (Incomplete Cholesky) – неполная факторизация Холесского
ILU (Incomplete LU) – неполная LU факторизация
Index Sets – индексные последовательности
Jacobi – метод Якоби
KSP (Krylov Subspace Methods) – методы подпространств Крылова
LAPACK – библиотека программ линейной алгебры
LU (Low-Upper) – разложение матрицы на нижнюю или верхнюю треугольные матрицы
Matrices – матрицы

- **Block Compressed Sparse Row (BAIJ)** – блочная разреженная сжатая по строкам матрица
- **Block Diagonal (Bdiag)** – блочная диагональная матрица
- **Compressed Sparse Row (AIJ)** – разреженная сжатая по строкам матрица
- **Dense –** плотная матрица

PC (Preconditioners) – переобусловливатели
Richardson – метод Ричардсона
SLES (Linear Equations Silvers) – решатели систем линейных алгебраических уравнений
SNES (Nonlinear Equotions Solvers) – решатели систем нелинейных алгебраических уравнений
SOR (Successive Overrelaxation) - метод последовательной верхней релаксации
SSOR (Symmetric Successive Overrelaxation) – метод симметричной последовательной верхней релаксации
Stride – шаг по индексу

УКАЗАТЕЛЬ ФУНКЦИЙ РАЗДЕЛА 3

ВЕКТОРНЫЕ ФУНКЦИИ

77. VecScatterBegin(Vecx,Vecy,INSERT_VALUES, ATTER_FORWARD,
VecScatter ctx); 125
78. VecScatterEnd(Vec x,Vec y,INSERT_VALUES,
SCATTER_FORWARD,VecScatter ctx); 125
79. VecScatterDestroy(VecScatter ctx); 125
80. VecScatterBegin(Vec y,Vec x,INSERT_VALUES,
SCATTER_REVERSE,VecScatter ctx); 126
81. VecScatterEnd(Vec y,Vec x,INSERT_VALUES,
SCATTER_REVERSE,VecScatter ctx); 126
82. VecCreateGhost(MPI_Comm comm,int n,int N,int nghost, int *ghosts,Vec *vv);
128
83. VecCreateGhostWithArray(MPI_Comm comm,int n,int N,int nghost,
int *ghosts,Scalar *array,Vec *vv); 128

МАТРИЧНЫЕ ФУНКЦИИ

1. MatCreate(MPI_Comm comm,int m,int n,int M,int N,Mat *A); 130
2. MatSetValues(Mat A,int m,int *im,int n,int *in,Scalar
*values,INSERT_VALUES); 130
3. MatSetValues(Mat A,int m,int *im,int n,int *in,Scalar *values, ADD_VALUES);
130
4. MatSetOption(Mat A,MAT_COLUMN_ORIENTED); 131
5. MatSetValuesBlocked(); 131
6. MatSetOption(Mat A,MAT_ROWS_SORTED); 131
7. MatSetOption(Mat A,MAT_COLUMNS_SORTED); 131
8. MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY); 131
9. MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY); 131
10. MatGetOwnershipRange(Mat A,int *first_row,int *last_row); 131
11. MatSetOption(Mat mat,MAT_NO_NEW_NONZERO_LOCATIONS); 132
12. MatCreateSeqAIJ(PETSC_COMM_SELF,int m,int n,int nz,int *nnz,Mat*A); 132
13. MatCreateMPIAIJ(MPI_Comm comm,int m,int n,int M,int N,int d_nz,*d_nnz,
int o_nz,int *o_nnz,Mat *A); 135
14. MatCreateSeqDense(PETSC_COMM_SELF,int m,int n,Scalar *data,Mat *A); 137
15. MatCreateMPIDense(MPI_Comm comm,int m,int n,int M,int N,
Scalar*data, Mat *A); 138
16. MatMult(Mat A,Vec x,Vec y); 138
17. VecCreateMPI(MPI_Comm comm,n,N,&x); 138
18. MatMultTranspose(Mat A,Vec x,Vec y); 138
19. MatMultAdd(Mat A,Vec x,Vec y,Vec w); 138
20. MatMultTransAdd(Mat A,Vec x,Vec y,Vec w); 139
21. MatView(Mat mat,PETSC_VIEWER_STDOUT_WORLD); 139
22. MatView(Mat mat,PETSC_VIEWER_DRAW_WORLD); 139
23. MatView(Mat mat,PetscViewer viewer); 139
24. VieverDrawOpenX(); 139
25. MatCreateShell(MPI_Comm comm,int m,int n,int M,int N,
void *ctx,Mat *mat); 140
26. UserMult(Mat mat,Vec x,Vec y); 140

27. MatShellSetOperation(Mat mat,MatOperation MATOP_MULT,
(void(*)()) int (*UserMult)(Mat,Vec,Vec)); 140
28. MatZeroEntries(Mat A); 141
29. MatZeroRows(Mat A,IS rows,Scalar *diag_value); 141
30. MatConvert(Mat mat,MatType newtype,Mat *M); 141
31. MatGetRow(Mat A,int row, int *ncols,int **cols,Scalar **vals); 142
32. MatRestoreRow(Mat A,int row,int *ncols,int **cols,Scalar **vals); 142

SLES

1. SLESCreate(MPI_Comm comm,SLES *sles); 143
2. SLESSetOperators(SLES sles,Mat Amat,Mat Pmat,MatStructureflag); 143
3. SLESSetFromOptions(SLES sles); 144
4. SLESSolve(SLES sles,Vec b,Vec x,int *its); 144
5. SLESDestroy(SLES sles); 144
6. SLESSetUp(SLES sles,Vec b,Vec x); 144
7. SLESGetPC(SLES sles,PC *pc); 144
8. SLESGetKSP(SLES sles,KSP *ksp); 144
9. KSPSetType(KSP ksp,KSPType method); 145
10. KSPRichardsonSetScale(KSP ksp,double damping_factor); 146
11. KSPChebychevSetEigenvalues(KSP ksp,double emax,double emin); 146
12. KSPGMRESSetRestart(KSP ksp,int max_steps); 146
13. KSPGMRESSetOrthogonalization(KSP ksp,
SPGMRESModifiedGramSchmidtOrthogonalization); 146
14. KSPGMRESSetOrthogonalization(KSP ksp,
SPGMRESUnmodifiedGramSchmidtOrthogonalization); 146
15. KSPCGSetType(KSP ksp,KSPCGType KSP_CG_SYMMETRIC); 146
16. KSPSetInitialGuessNonzero(KSP ksp); 146
17. KSPSetPreconditionerSide(KSP ksp,PCSide PC_RIGHT); 147
18. KSPSetUsePreconditionedResidual(KSP ksp); 148
19. KSPSetTolerances(KSP ksp,double rtol,double atol,double dtol,int maxits); 148
20. KSPSetConvergenceTest(KSP ksp,int (*test)(KSP ksp,int it,double rnorm,
PConvergedReason *reason,void *ctx), void *ctx); 149
21. KSPSetMonitor(KSP ksp,int (*mon)(KSP ksp,int it,
double rnorm,void *ctx), void *ctx,int (*mondestroy)(void *)); 149
22. PetscObjectGetComm ((PetscObject)ksp,MPI_Comm*comm); 150
23. KSPDefaultMonitor(KSP,int,double, void *); 150
24. KSPSingularValueMonitor(KSP,int,double, void *); 150
25. KSPTrueMonitor(KSP,int,double, void *); 150
26. PetscDrawLG lg; 150
27. KSPLGMonitorCreate(char *display,char *title,int x, int y,intw,int h,
PetscDrawLG *lg); 150
28. KSPSetMonitor(KSP ksp,KSPLGMonitor,lg,0); 150
29. KSPLGMonitorDestroy(PetscDrawLG lg); 150
30. KSPComputeEigenvalues(KSP ksp, int n,double *realpart,double
*complexpart,int neig); 151

31. KSPComputeEigenvaluesExplicitly(KSP ksp, int n,double *realpart,
double *complexpart); 151
32. KSPGetSolution(KSP ksp,Vec *x); 152
33. KSPGetRhs(KSP ksp,Vec *rhs); 152
34. KSPBuildSolution(KSP ksp,Vec w,Vec *v); 152
35. KSPBuildResidual(KSP ksp,Vec t,Vec w,Vec *v); 152
36. PCSetType(PC pc,PCType method); 152
37. PCILUSetLevels(PC pc,int levels); 153
38. PCILCCSetLevels(PC pc,int levels); 153
39. PCILUSetReuseOrdering(PC pc,PetscTruth flag); 153
40. PCILUSetUseDropTolerance(PC pc,double dt,int dtcount); 153
41. PCILUDTSetReuseFill(PC pc,PetscTruth flag); 153
42. PCILUSetUseInPlace(PC pc); 153
43. PCILUSetAllowDiagonalFill(PC pc); 153
44. MatCreateMPIRowbs(MPI_Comm comm,int m,int M,int nz,int *nnz,Mat *A); 155
45. MatSetOption(Mat mat,MAT_SYMMETRIC); 155
46. PCSORSetOmega(PC pc,double omega); 156
47. PCSORSetIterations(PC pc,int its); 156
48. PCSORSetSymmetric(PC pc,MatSORType type); 156
49. PCLUSetUseInPlace(PC pc); 156
50. PCBJacobiGetSubSLES(PC pc,int *n_local,int *first_local,SLES **subsles); 158
51. PCASMGetSubSLES(PC pc,int *n_local,int *first_local,SLES **subsles); 158
52. PCBJacobiSetTotalBlocks(PC pc,int blocks,int *size); 158
53. PCASMSetTotalSubdomains(PC pc,int n,IS *is); 158
54. PCASMSetType(PC pc,PCASMType type); 158
55. PCBJacobiSetLocalBlocks(PC pc,int blocks,int *size); 159
56. PCASMSetLocalSubdomains(PC pc,int N,IS *is); 159
57. PCASMSetOverlap(PC pc,int overlap); 159
58. PCShellSetApply(PC pc,int (*apply)(void *ctx,Vec,Vec),void *ctx); 159
59. PCShellSetSetUp(PC pc,int (*setup)(void *ctx)); 159
60. PCSetType(pc,PCCOMPOSITE); 160
61. PCCompositeAddPC(pc,type1); 160
62. PCCompositeAddPC(pc,type2); 160
63. PCCompositeSetType(PC pc,
PCCompositeType PC_COMPOSITE_MULTIPLICATIVE); 160
64. PCCompositeSetUseTrue(PC pc); 160
65. PCCompositeGetPC(PC pc,int count,PC *subpc); 161
66. PC subpc; 161
67. PCCompositeGetPC(pc,0,&subpc); 161
68. PCILUSetFill(subpc,1); 161
69. PCSetType(PC pc,PCSLES PCSLES); 161
70. PCSLESGetSLES(pc,&sles); 161
71. PCSLESSetUseTrue(PC pc); 161

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

ЛИТЕРАТУРА

1. Сайт http://www.mcs.anl.gov/mpi (Argonne National Laboratory, США).
2. Сайт http://www.parallel.ru (НИВЦ МГУ).
3. Сайт http://www.cluster.bsu.by (Белгосуниверситет, Минск).
4. *В. Воеводин*, *Вл. Воеводин*. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 609 с.
5. *Шпаковский Г. И.* Организация параллельных ЭВМ и суперскалярных процессоров: Учеб. пособие. Мн.: Белгосуниверситет, 1996. 284 с.
6. *Буза М. К.* Введение в архитектуру компьютеров: Учеб. пособие. Мн.: БГУ, 2000. 253 с.
7. Сайт http://www.top500.org
8. *Шпаковский Г. И.*, *Серикова Н. В.* Программирование для многопроцессорных систем в стандарте MPI. Мн.: БГУ, 2002. 323 с.
9. *Лацис А. О.* Как построить и использовать суперкомпьютер. М.: Бестселлер. 2003. 240 с.
10. *Олифер В. Г.*, *Олифер Н. А.* Компьютерные сети. Принципы, технологии, протоколы:Учебник для вузов. 2-е изд. СПб.: Питер, 2003. 864 с.
11. *Корнеев В. В.* Параллельные вычислительные системы. М.: Нолидж, 1999. 320 с.
12. *Гергель В. П.* Оценка эффективности параллельных вычислений для Intel-процессорных вычислительных кластеров. // Материалы междунар. науч.-практ. семинара «Высокопроизводительные параллельные вычисления на кластерных системах». / Под ред. проф. Стронгина Р. Г. Нижний Новгород: Изд-во Нижегород. ун-та, 2002.
13. *Немнюгин С. А.*, *Стесик О. Л.* Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ- Петербург, 2002. 400 с.
14. *Ортега Дж.* Введение в параллельные и векторные методы решения линейных систем: Пер. с англ. М.: Мир, 1991.367 с.
15. *Мулярчик С. Г.* Численные методы: Конспект лекций. Мн., БГУ, 2001.127 с.
16. *Крылов В. И.*, *Бобков В. В.*, *Монастырный П. И.* Вычислительные методы. М.: Наука, 1977 г. 400 с.
17. *Вержбицкий В. М.* Численные методы. М.: Высш. школа, 2000 г. 268 с.
18. *Самарский А. А.* Введение в численные методы. М.: Наука, 1987 г. 286 с.
19. *Демидович Б. П.*, *Марон И. А.* Основы вычислительной математики. М.: Наука, 1966 г. 664 с.

ОГЛАВЛЕНИЕ