Учебник по программированию для многопроцессорных систем

Содержание

Предисловие

Идея параллельной обработки данных не нова. Можно считать, что она возникла еще на заре человеческой цивилизации, когда оказалось, что племя может успешно бороться за выживание, если каждый его член выполняет свою часть общей работы. А представьте себе учреждение, где один человек работает в качестве директора, секретаря директора, курьера, успевает поработать у станка, отремонтировать электропроводку, а ночью еще и охраняет свой объект. Выглядит это почти нелепо. Можно допустить, что такая организация выполнит определенную работу, но времени на это уйдет очень много. А ведь в таком режиме, фактически, работает обычный последовательный компьютер, справляясь с решением поставленных перед ним задач только благодаря быстрой работе центрального процессора. Совсем другое дело, когда в организации есть несколько сотрудников, которые работают одновременно, и каждый из них решает свою часть общей задачи. Можно считать, что режим параллельной обработки данных является наиболее естественным "режимом" существования и человеческого общества.

Неудивительно, что еще на заре компьютерной эры, примерно в середине прошлого века, конструкторы электронно-вычислительных машин задумались над возможностью применения параллельных вычислений в компьютерах. Ведь увеличение быстродействия только за счет совершенствования электронных компонентов компьютера — достаточно дорогой способ, который, к тому же, сталкивается с ограничениями, налагаемыми физическими законами. Так параллельная обработка данных и параллелизм команд были введены в конструкцию компьютеров и сейчас любой пользователь "персоналки", возможно, сам того не зная, работает на параллельном компьютере. Впрочем, обычному пользователю персонального компьютера и не надо об этом знать.

Вместе с тем, достаточно часто приходится сталкиваться с такими задачами, которые, представляя немалую ценность для общества, не могут быть решены с помощью относительно медленных компьютеров офисного или домашнего класса. Единственная надежда в этом случае возлагается на компьютеры с большим быстродействием, которые принято называть суперкомпьютерами. Только машины такого класса могут справиться с обработкой больших объемов информации. Это могут быть, например, статистические данные или результаты метеорологических наблюдений, финансовая информация. Иногда скорость обработки имеет решающее значение. В качестве примера можно привести составление прогноза погоды и моделирование климатических изменений. Чем раньше предсказано стихийное бедствие, тем больше возможностей подготовиться к нему. Важной задачей является моделирование лекарственных средств, расшифровка генома человека. А томография, в том числе и медицинская? А разведка месторождений нефти и газа? Примеров можно привести много.

В середине 60-х годов прошлого века появились первые суперкомпьютеры, это были

параллельный 64-процессорный компьютер ILLIAC-IV и векторный компьютер CDC 6500. Последний стал дебютом легендарного конструктора высокопроизводительных вычислительных систем Сеймура Крея.

Сеймур Крей основал знаменитую фирму Cray, которая и сейчас выпускает одноименные суперкомпьютеры.

Из современных проектов следует упомянуть ASCI (Accelerated Strategic Computer Initiative) — проект, в рамках которого создаются вычислительные системы с рекордной производительностью. В качестве примера можно привести суперкомпьютеры ASCI Red и ASCI White, которые вошли в рейтинг наиболее мощных вычислительных систем мира "Top 500 Supercomputers". Правительство США вкладывает в этот проект большие деньги, поскольку одним из применений таких систем является имитационное моделирование ядерного оружия, которое рассматривается в качестве альтернативы реальным испытаниям.

Вспоминая историю развития суперкомпьютерных технологий, нельзя не сказать о вкладе тогда еще советских ученых и конструкторов. Авторам довелось поработать на таких машинах высочайшего класса, как БЭСМ-6 и "Эльбрус".

Суперкомпьютеры являются дорогими, элитными машинами, стоимость которых может достигать десятков миллионов долларов. Одним из решений проблемы доступа к дорогостоящим суперкомпьютерным ресурсам является создание центров, обслуживающих несколько организаций. Так, в США имеется около десятка крупных суперкомпьютерных центров. Имеются такие центры и в России, наиболее крупные из них расположены в Москве и Санкт-Петербурге.

Примерно в середине 80-х годов прошлого века ученые заинтересовались возможностью использования кластерных систем в качестве относительно недорогих заменителей суперкомпьютеров. Благодаря развитию сетевых технологий, появилась возможность соединять вместе несколько рабочих станций, в качестве которых могут использоваться обычные персональные компьютеры. Ну а чем такая система не многопроцессорный вычислительный комплекс? Было разработано необходимое программное обеспечение, которое стало распространяться бесплатно. Работало это программное обеспечение в среде операционной системы Linux, которая тоже является бесплатной, оставаясь в то же время чрезвычайно гибкой и надежной. Таким образом, основная часть программного обеспечения кластера оказывается бесплатной, что значительно снижает суммарную стоимость всей системы. Оказалось, что в результате можно построить многопроцессорный комплекс, который лишь в 2—3 раза уступает по быстродействию суперкомпьютеру, но дешевле его в несколько сотен раз. Такие системы, которые получили название Linux-ферм, быстро завоевали популярность. Этот кластер активно используют ученые университета, физики, химики, математики для решения сложнейших научных задач.

Следует отметить, что хотя исследования в области применения кластеров на Западе ведутся уже достаточно давно, в России интерес к ним появился сравнительно недавно. Отрадно, что "суперкомпьютинг" получил реальную поддержку со стороны государства. Ведь суперкомпьютерные ресурсы недаром относят к числу стратегических. Принята государственная программа развития суперкомпьютерных технологий, созданы суперкомпьютерные центры, ведется обучение специалистов. К сожалению, ощущается недостаток учебной литературы, посвященной вопросам параллельного программирования. Изданы книги по общим вопросам архитектуры параллельных и векторных вычислительных систем, по параллельным алгоритмам, а вот описания средств разработки параллельных программ нет... Авторы надеются, что их скромные усилия позволят восполнить этот недостаток. Наша книга рассчитана на программиста, уже имеющего определенный опыт разработки обычных, последовательных программ.

Немного о структуре книги.

В *главе 1* рассматривается архитектура высокопроизводительных ЭВМ. Здесь мы знакомим читателя с основными концепциями архитектуры высокопроизводительных вычислительных систем, такими как конвейеры данных и команд, векторная обработка данных. Рассматриваются особенности устройства памяти, подсистемы коммуникаций. Общее понимание принципов работы высокопроизводительной вычислительной системы полезно разработчику параллельных программ.

В *главе 2* речь идет об особенностях программирования параллельных и векторных вычислений. Здесь мы уделим внимание технологии разработки параллельных алгоритмов, познакомимся с различными моделями программирования. Эта глава завершает часть книги, посвященную общим вопросам параллельного программирования.

Остальная часть содержит описание наиболее распространенных программных средств разработки параллельных программ.

Глава 3 служит введением в одну из реализаций спецификации MPI (Message Passing Interface) -MPICH, которая основана на модели передачи сообщений. Рассматриваются общая организация и структура MPICH, организация обмена данными. Вводится концептуальное для MPI понятие коммуникатора. Описывается структура MPI-программы, компиляция и запуск MPI-программ.

В *главе 4* излагаются средства MPICH, с помощью которых организуется обмен сообщениями. Рассматриваются различные режимы двухточечного обмена, коллективный обмен, управление группами процессов. Даются описания подпрограмм библиотеки MPICH, приводятся примеры программ.

В *главе 5* рассматриваются более сложные вопросы программирования с

использованием MPI. Это конструирование производных типов данных, топологии и другие вопросы.

В *главе 6* даются основы другой популярной системы параллельного программирования — PVM (Parallel Virtual Machine). Рассматривается архитектура PVM, настройка системы, работа с PVM-консолью. Дано описание основных подпрограмм системы. Приведен сравнительный обзор MPI и PVM, который позволит читателю при необходимости выбрать оптимальную для его задач систему параллельного программирования.

Глава 7 содержит описание подпрограмм библиотек PVM для обмена сообщениями, управления процессами и виртуальной машиной. Разбираются такие вопросы, как динамическое изменение конфигурации виртуальной машины, буферизация данных и некоторые другие.

В последней, *главе 8* дается описание языка Высокопроизводительный FORTRAN, а также необходимые для понимания материала сведения о языке FORTRAN-90. Приводятся примеры программ.

В *приложениях* читатель найдет описание средств отладки и мониторинга параллельных программ, рекомендации по созданию и наладке своего собственного параллельного кластера и краткий обзор ресурсов Интернета, посвященных высокопроизводительным вычислениям.

ГЛАВА 1.

Архитектура высокопроизводительных ЭВМ

Прежде чем мы перейдем к изучению методов и средств параллельного программирования, полезно познакомиться с некоторыми особенностями устройства высокопроизводительных вычислительных систем. Преимуществом использования языков программирования высокого уровня является универсальность программ и их простая переносимость между различными компьютерами (разумеется, если на этих компьютерах имеется необходимое программное обеспечение, прежде всего, трансляторы). Следует учитывать, что эффективность выполнения параллельных программ определяется не только аппаратной частью, но и способностью транслятора генерировать эффективный исполняемый код. Оба эти фактора взаимосвязаны и порой сложно определить, какой из них имеет решающее значение.

Обычно, программист полагается на эффективность транслятора, считая, что при грамотном программировании сгенерированный транслятором исполняемый код будет обладать хорошими показателями по быстродействию; в этом случае отпадает необходимость в применении языков низкого уровня (ассемблеров), требующих глубоких знаний архитектуры процессора.

Но, бывает и так, что программисту все же требуется знание устройства компьютера, принципов и даже некоторых деталей его работы. Можно привести следующий пример. При разработке программ для параллельных ЭВМ используются специализированные библиотеки, позволяющие организовать обмен данными между отдельными подзадачами и их синхронизацию. Для эффективной организации обмена данными необходимо знать, как происходит пересылка информации между процессорами, какова ее скорость и т. д.

В вычислительной технике используются три термина, связанные с устройством электронно-вычислительной машины:

1. *Архитектура компьютера* — это описание основных компонентов компьютера и их взаимодействия. Можно считать, что архитектура — это те атрибуты вычислительной системы, которые видны программисту: набор команд, разрядность машинного слова, механизмы ввода/вывода, методы адресации. Можно дать и другое определение: архитектура — это внутренняя структура системы или микропроцессора, которая определяет их функциональные возможности и быстродействие.

2. *Организация компьютера* — это описание конкретной реализации архитектуры, ее воплощения "в железе". Оно включает перечень используемых сигналов управления, интерфейсов, технологий памяти, реализацию арифметических, логических и других операций. Так, умножение может быть реализовано на аппаратном уровне или посредством многократного повторения операции суммирования. Ряд суперкомпьютеров Cray, например, имеют сходную архитектуру. С точки зрения программиста у них одинаковое число внутренних регистров, используемых для временного хранения данных, одинаковый набор машинных команд, одинаковый формат представления данных. Организация же компьютеров Cray разных моделей может существенно различаться. У них может быть разное число процессоров, разный размер оперативной памяти, разное быстродействие и т. д.

3. *Схема компьютера* — детальное описание его электронных компонентов, их соединений, устройств питания, охлаждения и др. Программисту довольно часто требуется знание архитектуры компьютера, реже — его организации и никогда — схемы компьютера. Действительно, зачем специалисту, который занимается, например, расчетом прочностных свойств новой модели автомобиля, знать, какая марка вентилятора охлаждает процессор компьютера во время выполнения его программы?

Классификация Флинна

Одной из наиболее известных схем классификации компьютерных архитектур является *таксономия Флинна,* предложенная Майклом Флинном в 1972 году. В ее основу положено описание работы компьютера с потоками команд и данных. В классификации Флинна имеется четыре класса архитектур:

1. SISD (Single Instruction Stream — Single Data Stream) — один поток команд и один поток данных.

2. SIMD (Single Instruction Stream — Multiple Data Stream) — один поток команд и несколько потоков данных.

3. MISD (Multiple Instruction Stream — Single Data Stream) — несколько потоков команд

и один поток данных.

4. MIMD (Multiple Instruction Stream — Multiple Data Stream) — несколько потоков команд и несколько потоков данных.

Рассмотрим эту классификацию более подробно.

SISD-компьютеры

SISD-компьютеры (рис. 1.1) — это обычные последовательные компьютеры, выполняющие в каждый момент времени только одну операцию над одним элементом данных. Большинство современных персональных ЭВМ принадлежат именно этой категории.

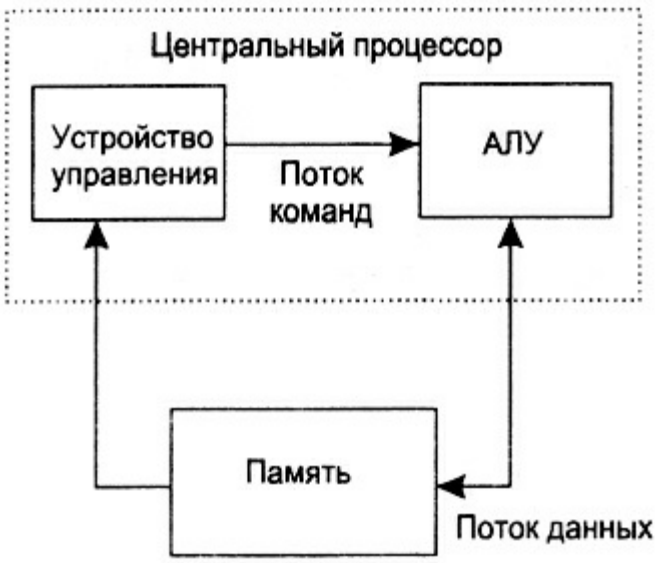


Рис. 1.1. *Схема SISD-компьютера*

Многие современные вычислительные системы включают в свой состав несколько процессоров, но каждый из них работает со своим независимым потоком данных, относящимся к независимой программе. Такой компьютер является, фактически, набором SISD-машин, работающих с независимыми множествами данных.

SIMD-компьютеры

SIMD-компьютеры (рис. 1.2 и 1.3) состоят из одного командного процессора (управляющего модуля), называемого *контроллером,* и нескольких модулей обработки данных, называемых *процессорными элементами* (ПЭ). Количество модулей обработки данных таких машин может быть от 1024 до 16 384. Управляющий модуль принимает, анализирует и выполняет команды. Если в команде встречаются данные, контроллер рассылает на все ПЭ команду, и эта команда выполняется либо на нескольких, либо на всех процессорных элементах. Процессорные элементы в SIMD-компьютерах имеют относительно простое устройство, они содержат

арифметико-логическое устройство (АЛУ), выполняющее команды, поступающие из устройства управления (УУ), несколько регистров и локальную оперативную память. Одним из преимуществ данной архитектуры считается эффективная реализация логики вычислений. До половины логических команд обычного процессора связано с управлением процессом выполнения машинных команд, а остальная их часть относится к работе с внутренней памятью процессора

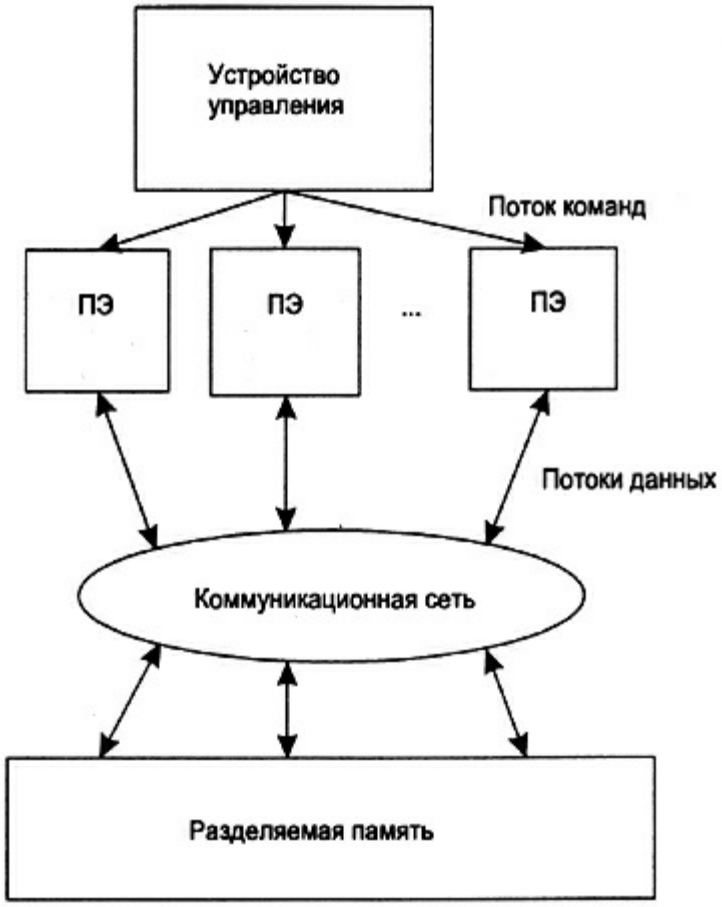


Рис. 1.2. *Схема SIMD-компьютера с разделяемой памятью*

Рис. 1.3. *Схема SIMD-компьютера с распределенной памятью*

и выполнению арифметических операций. В SIMD-компьютере управление выполняется контроллером, а "арифметика" отдана процессорным элементам. Подклассом SIMD-компьютеров являются векторные компьютеры. Пример такой вычислительной системы — Hitachi S3600.

Другой пример SIMD-компьютера — *матричные процессоры* (Array Processor). В качестве примера можно привести вычислительную систему Thinking Machines CM-2, где 65 536 ПЭ связаны между собой сетью коммуникаций с топологией "гиперкуб". Часто компьютеры с SIMD-архитектурой специализированы для решения конкретных задач, допускающих матричное представление. Это, например, могут быть задачи обработки изображений, где каждый модуль обработки данных работает на получение одного элемента конечного результата.

MISD-компьютеры

Вычислительных машин такого класса мало. Один из немногих примеров - *систолический массив* процессоров, в котором процессоры находятся в узлах регулярной решетки. Роль ребер в ней играют межпроцессорные соединения, все ПЭ управляются общим тактовым генератором. В каждом цикле работы любой ПЭ получает данные от своих соседей, выполняет одну команду и передает результат соседям. На рис. 1.4 дана схема фрагмента систолического массива.

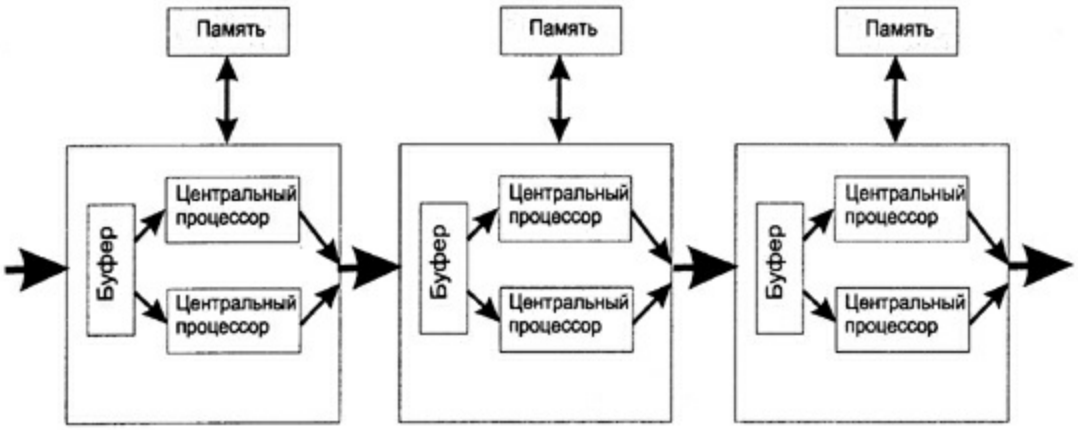


Рис. 1.4. *Схема MISD-компьютера*

MIMD-компьютеры

Этот класс архитектур (рис. 1.5 и 1.6) наиболее богат примерами успешных реализаций. В него попадают симметричные параллельные вычислительные системы, рабочие станции с несколькими процессорами, кластеры рабочих станций и т. д. Довольно давно появились компьютеры с несколькими независимыми процессорами, но вначале на них был реализован только принцип параллельного исполнения заданий, т. е. на разных процессорах одновременно выполнялись независимые программы. Разработке первых компьютеров для параллельных вычислений были посвящены проекты под условным названием CM* и C.MMP в университете Карнеги (США). Технической базой для этих проектов были процессоры DEC PDP-11. В начале 90-х годов прошлого века именно MIMD-компьютеры вышли в лидеры на рынке высокопроизводительных вычислительных систем.

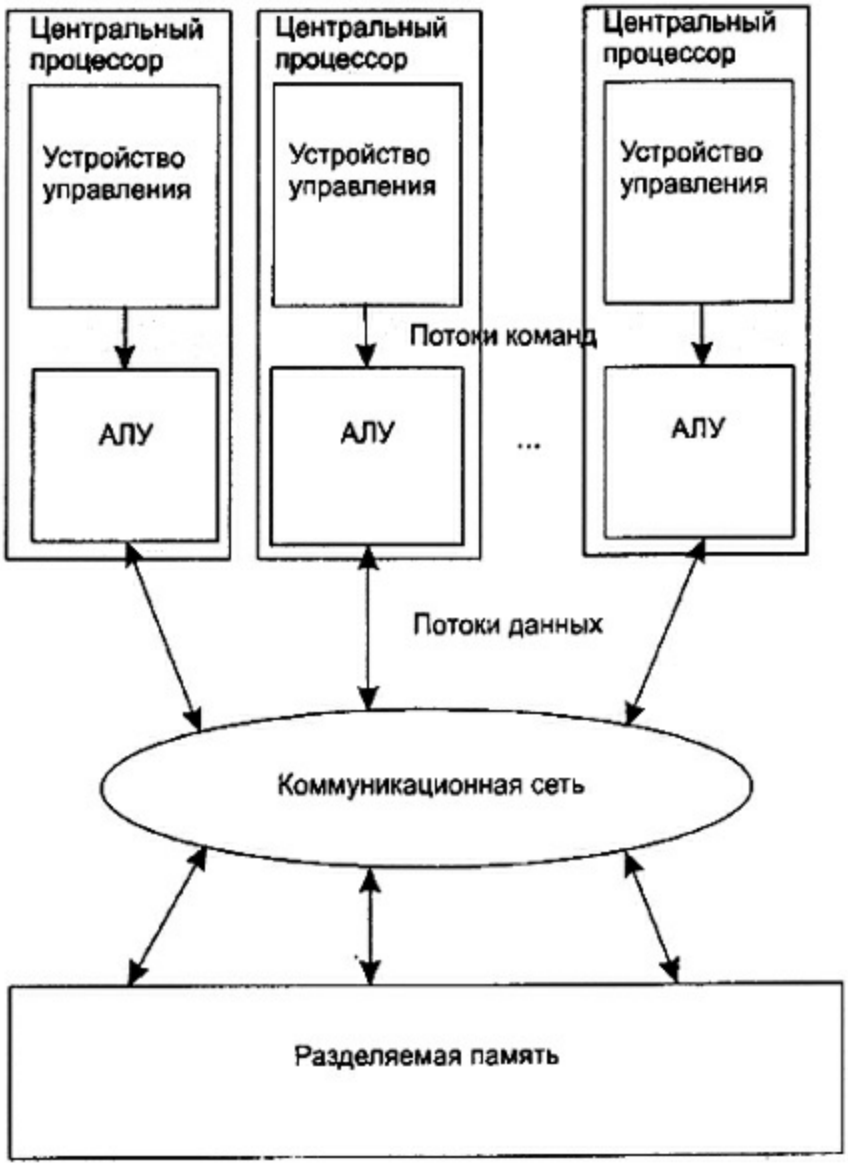


Рис. 1.5. *Схема MIMD-компьютера с разделяемой памятью*

Имеются и гибридные конфигурации, в которых, например, объединены несколько SIMD-компьютеров, в результате чего получается MSIMD-компьютер, позволяющий создавать виртуальные конфигурации, каждая из которых работает в SIMD-режиме.

Рис. 1.6. *Схема MIMD-компьютера с распределенной памятью*

Классификация Флинна не дает исчерпывающего описания разнообразных архитектур MIMD-машин, порой существенно отличающихся друг от друга. Например, существуют такие подклассы MIMD-компьютеров, как системы с разделяемой памятью и системы с распределенной памятью. Системы с разделяемой памятью могут относиться по классификации Флинна как к MIMD, так и к SIMD-машинам. То же самое можно сказать и о системах с распределенной памятью.

Развитием концепции MIMD-архитектуры с распределенной памятью является *распределенная обработка,* когда вместо набора процессоров в одном корпусе используются компьютеры, связанные достаточно быстрой сетью. Концептуального отличия от MIMD-архитектуры с распределенной памятью нет, а особенностью является медленное сетевое соединение.

Традиционная архитектура фон Неймана

Традиционная фон-неймановская архитектура компьютера представлена на рис. 1.7. Она основана на следующих принципах:

- программа хранится в компьютере;
- программа во время выполнения и необходимые для ее работы данные находятся в оперативной (главной) памяти;
- имеется *арифметико-логическое устройство,* выполняющее арифметические и логические операции с данными;
- имеется *устройство управления,* которое интерпретирует команды, выбираемые из памяти, и выполняет их;
- *устройства ввода и вывода* (ВВ) используются для ввода программ и данных и для вывода результатов расчетов. Работают под управлением УУ.

Рис. 1.7. *Схема компьютера фон Неймана*

Устройство управления и арифметико-логическое устройство (АЛУ) вместе составляют *центральный процессор* (ЦП). Можно считать, что все остальные функциональные блоки компьютера обслуживают АЛУ, а оно выполняет основную работу. Арифметические операции выполняются с целыми и вещественными числами, для представления которых используется двоичный формат. Для отрицательных значений применяется дополнительный код, что позволяет упростить реализацию арифметических и логических операций. Управляющее же устройство управляет работой остальных функциональных узлов компьютера.

В состав "традиционного" фон-неймановского компьютера входит один ЦП, основной задачей которого является выполнение машинных команд, выбираемых из оперативной памяти по очереди, одна за другой. Устройство управления

интерпретирует очередную команду, которая входит в набор команд процессора, включающих арифметические и логические операции, операции пересылки данных и некоторые другие. Затем выполняется выборка данных, их обработка и запись результата выполнения в оперативную память. ЦП имеет набор регистров — устройств для временного хранения промежуточных результатов и данных, необходимых для выполнения команд (операндов).

Таким образом, основными компонентами компьютера являются:

- центральный процессор и оперативная память, образующие ядро системы;
- вторичная ("внешняя") память и устройства ввода/вывода, образующие "периферию";
- коммуникации между компонентами системы, осуществляемые посредством шин.

Регистры

Регистры центрального процессора представляют собой вспомогательную память, находящуюся на вершине иерархической организации памяти (об иерархии памяти мы поговорим чуть позже). Количество регистров различается в процессорах разного типа. Программисту доступны:

- регистры общего назначения (General Purpose Registers);
- регистры данных (Data Registers);
- регистры адреса (Address Registers);
- регистр кодов условий (Condition Codes Register).

Имеются также управляющие регистры и регистры состояния:

- счетчик команд (Program Counter);
- регистр декодирования команд (Instruction Register);
- регистр адресации памяти (Memory Addressing Register);
- регистр буфера памяти (Memory Buffer Register).

Выполнение команды

Выполнение команды состоит из двух этапов:

- выборка;
- выполнение.

Первый шаг — выборка — выполняется следующим образом. Вначале из счетчика команд выбирается адрес очередной команды, при этом значение, содержащееся в счетчике команд, увеличивается. Адрес команды помещается в регистр адресации памяти и передается в адресную шину. УУ считывает данные из оперативной памяти.

Результат считывания помещается в шину данных, копируется в регистр буфера памяти, а затем в регистр декодирования команд.

При выборке данных анализируется регистр декодирования команд и производится выборка данных. Младшие биты регистра буфера памяти передаются в регистр адресации памяти, затем УУ формирует запрос к памяти и результат выполнения этого запроса (адрес или операнд) пересылается в регистр буфера памяти.

Второй шаг — выполнение. Во время этого цикла происходит пересылка данных между центральным процессором и оперативной памятью и между центральным процессором и модулем ввода/вывода; выполняются арифметические и логические операции над данными. При этом, если есть переходы, может изменяться последовательность выполнения команд.

Набор команд процессора

Центральный процессор выполняет машинные команды из набора, определенного для данной разновидности процессоров. Для представления этих команд используется двоичный формат. Каждая машинная команда включает код операции, ссылку на операнды, адрес записи результата операции и ссылку на следующую по порядку команду.

Набор команд процессора характеризуется:

- типами операндов;
- форматом команд;
- допустимыми командами;
- количеством адресов в команде;
- доступом к регистрам;
- режимами адресации.

Большинство машинных команд можно разделить на четыре категории:

- арифметические и логические операции;
- операции пересылки данных между оперативной памятью и регистрами центрального процессора;
- операции ввода/вывода;
- команды управления (проверки, ветвления).

Пример последовательности машинных команд для вычисления арифметического выражения и оператора присваивания z := (x + Y) * 7 приведен в табл. 1.1.

Таблица 1.1. *Пример последовательности машинных команд*

Адрес команды	Команда
00001000	00001 01110001 111
	Move адрес Y Регистр 7
00001001	00011 01110000 111
	Add адрес X Регистр 7
00001010	00101 00000011 111
	Mul операнд 3 Регистр 7
00001011	00010 01110010 111 Move адрес Z Регистр 7

Каждая команда состоит из последовательности элементарных шагов, которые в совокупности составляют цикл команды. Эти шаги являются более "мелкими" составными частями команды, чем упоминавшиеся ранее этапы выборки и выполнения. Управление выполнением команд реализуется как аппаратными средствами, так и средствами микропрограммирования.

Шины

В состав компьютера входят шины. *Шина* — это коммуникационная линия, соединяющая несколько (два или более) устройств. *Шины данных* используются для передачи данных. От ширины или разрядности шины зависит ее производительность, т. е. скорость передачи информации. *Адресная шина* определяет источник и приемник данных. *Шина управления* передает управляющую информацию — сигналы считывания и записи в память, запросы на прерывания, блокирующие сигналы, обеспечивает синхронизацию устройств. При подключении к одной шине нескольких устройств возникают задержки с передачей данных, поскольку в каждый момент времени воспользоваться шиной может только одно устройство, все остальные вынуждены в это время простаивать. Избавиться от таких задержек можно, применяя

несколько шин, т. е. распараллеливая процесс передачи информации.

Память

Основными компонентами подсистемы памяти компьютера являются:

- *оперативная (главная) память* — характеризуется высокой скоростью работы (малым временем выборки), сравнительно небольшим объемом и повышенной стоимостью;
- *внешняя (вторичная) память* — имеет большой объем, она дешевле, но характеризуется меньшей скоростью доступа.

В оперативной памяти компьютера размещается программа во время выполнения. Внешняя память используется для долговременного, постоянного хранения информации, в том числе программ, данных и т. д. В качестве устройств длительного хранения информации применяются жесткие диски, RAID-массивы, диски CD-ROM, магнитные ленты, дискеты и другие носители информации. Для выполнения программа загружается в оперативную память из внешней памяти. Важнейшими характеристиками памяти являются ее объем и скорость (время) доступа.

Оперативную память можно представлять себе состоящей из ячеек памяти — минимальных элементов, к которым можно обратиться из программы. Каждая ячейка имеет свой уникальный адрес. Имеются также буфер адресов (содержит адреса ячеек памяти, из которых производится считывание информации, или в которые производится запись), буфер данных (содержит данные для записи в ячейку памяти или для считывания из ячейки), а также декодер адреса и устройство управления памятью. Информация, находящаяся в оперативной памяти, пропадает при выключении компьютера, а информация, содержащаяся во внешней памяти, сохраняется и после выключения питания.

Для эффективной работы компьютера необходимо, чтобы операции с данными в оперативной памяти выполнялись со скоростью, сравнимой со скоростью работы центрального процессора. С другой стороны, известно, что скорость работы с памятью тем меньше, чем больше ее объем. Быстродействующая память стоит дороже, но для работы многих современных программ требуются большие объемы памяти. Особенно это относится к вычислительным программам, предназначенным для решения сложных задач. Возникает проблема, для решения которой используется иерархическая организация памяти. В такой иерархии можно выделить два уровня — маленькую, но быструю память, и медленную память большего объема. К быстрой памяти относятся регистры процессора, кэш-память первого и второго уровня, а к медленной — главная и внешняя память (рис. 1.8).

На вершине иерархии находятся регистры процессора со временем доступа порядка наносекунд. Емкость одного регистра обычно несколько десятков битов (например, 32

бита). Далее следует кэш-память первого уровня объемом до нескольких десятков килобайтов и временем доступа около десяти наносекунд. Следующий уровень иерархии — кэш-память второго уровня величиной несколько сот килобайтов и временем доступа в несколько раз большим, чем у кэш-памяти первого уровня. Оперативная память имеет объем несколько десятков и сотен мегабайтов и время доступа около сотни наносекунд. Затем следует внешняя память объемом в десятки гигабайтов и временем доступа в десятки миллисекунд.

Рис. 1.8. *Иерархия памяти компьютера*

Иерархическая организация памяти должна обеспечить доступность необходимых данных тогда, когда они понадобятся, причем с наибольшей скоростью доступа, которую может обеспечить только верхний уровень иерархии. Данные, находящиеся в регистрах, контролируются транслятором или программистом, если он программирует на ассемблере. Содержимое других уровней иерархии контролируется автоматически — кэш-памяти аппаратно, оперативной и внешней памяти операционной системой с участием аппаратуры.

Кэш-память — это быстрая память небольшого объема, содержащая информацию из оперативной памяти, которая использовалась недавно. Идея кэшпамяти основана на *принципе локальности.* Принцип локальности состоит в том, что если программа обратилась к какому-то элементу данных, этот элемент может вскоре понадобиться вновь. Это локальность во времени. Кроме того, вскоре могут потребоваться элементы данных, адреса которых мало отличаются от адреса данных, используемых в настоящий момент. Это локальность в пространстве.

Система сама решает, какую информацию следует сохранить в кэш-памяти. Для этого

используются разные алгоритмы. Схема взаимодействия кэшпамяти с главной памятью приведена на рис. 1.9.

Общая производительность вычислительной системы зависит от доли запросов к памяти, которая может быть обеспечена данными из кэш-памяти. В лучшем случае эта доля составляет несколько процентов. Распределение памяти и управление кэшированием возлагается на транслятор, программисту не надо об этом заботиться.

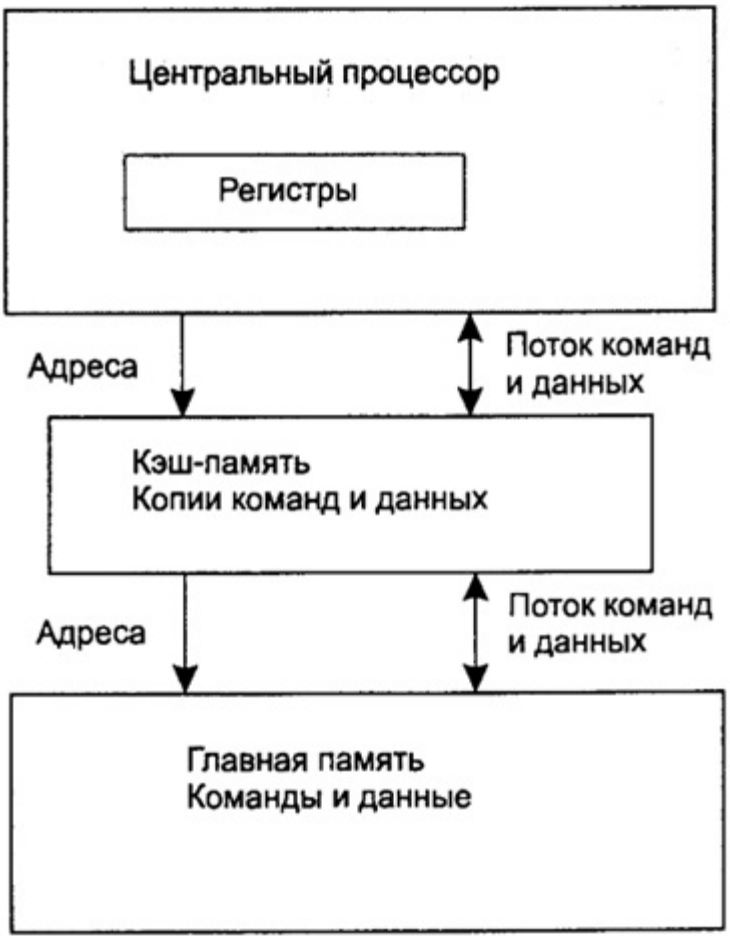


Рис. 1.9. *Взаимодействие между кэш-памятью и главной памятью*

Конструктивно кэш-память может выполняться по-разному. Она может быть общей для данных и для команд, но может быть и раздельной. Оба варианта имеют свои достоинства и свои недостатки. Так, первый вариант дешевле и позволяет сбалансировать потоки команд и данных. Второй вариант дороже, но эффективность его может быть выше благодаря параллелизму обработки потоков команд и данных.

В наборах команд современных процессоров появились команды записи данных из регистров в оперативную память, минуя кэш; чтения данных из памяти в регистры, минуя кэш; запись данных из памяти выборочно в кэш первого и второго уровня и др. Это позволяет оптимизировать работу с кэш-памятью.

Виртуальная память

Адресное пространство, которое требуется программам, часто превышает размер оперативной памяти. В этом случае в оперативной памяти может быть размещена только часть программы. Остальная ее часть находится на жестком диске. Для продолжения выполнения программы в оперативной памяти должна размещаться необходимая часть программы (сегмент команд и сегмент данных). В процессе выполнения программы приходится многократно загружать в оперативную память данные и выгружать часть программы на жесткий диск. Заботу об этом берет на себя операционная система. Единое адресное пространство, используемое программой в оперативной памяти и на жестком диске, называют *виртуальной памятью.*

Если виртуальный адрес команды или элемента данных относится к той части программы, которая размещается в оперативной памяти или в кэшпамяти, он преобразуется в физический адрес. Если же соответствующая часть программы находится на диске, она сначала загружается в оперативную память, а затем уже производится выборка.

Виртуальное адресное пространство делится на секции одинаковой величины, которые называются *страницами.* Выгрузка на диск и загрузка в память производятся страницами, процесс выгрузки страниц на диск именуется *свопингом* или *пейджингом* (постраничным свопингом). Программа состоит из большого числа страниц, но только малое их количество должно размещаться в оперативной памяти. Операционная система управляет перемещением страниц между жестким диском и оперативной памятью так, чтобы избежать ситуаций, когда центральный процессор обращается к странице, выгруженной на диск.

Виртуальный адрес состоит из номера страницы и смещения (адреса относительно начала страницы). При выполнении операций чтения или записи в оперативную память необходимо преобразовать виртуальный адрес в адрес физический (номер реальной ячейки оперативной памяти). Преобразование виртуального адреса выполняет *устройство управления памятью* (Memory Management Unit) с помощью таблицы страниц.

Устройства ввода/вывода

Человек или внешнее устройство взаимодействует с компьютером с помощью устройств ввода/вывода. Среди них: клавиатура, мышь, световое перо, джойстик, принтер, дисплей и др.

Основные элементы архитектуры высокопроизводительных вычислительных систем

Мы выяснили, что архитектура традиционных последовательных компьютеров основана на идеях Джона фон Неймана и включает в себя центральный процессор, оперативную память и устройства ввода/вывода. Последовательность команд

применяется к последовательности данных. Скорость работы такого компьютера определяется быстродействием его центрального процессора и временем доступа к оперативной памяти. Быстродействие центрального процессора может быть увеличено за счет увеличения тактовой частоты, величина которой зависит от плотности элементов в интегральной схеме, способа их "упаковки" и быстродействия микросхем оперативной памяти. И, тем не менее, традиционные однопроцессорные системы не могут обеспечить достаточное быстродействие для решения сложных задач. Для моделирования сложных систем в физике, экономике, биологии, технике, для обработки больших объемов информации требуется выполнение значительного объема вычислений.

Другие методы повышения быстродействия основаны на расширениях традиционной фон-неймановской архитектуры, включающих:

- конвейерную обработку данных и команд;
- использование процессоров с сокращенным набором команд (RISC-процессоров). В RISC-процессорах большая часть команд выполняется за 1—2 такта;
- использование суперскалярных процессоров;
- векторную обработку данных;
- использование процессоров со сверхдлинным командным словом;
- использование многопроцессорных конфигураций.

В этом разделе мы кратко рассмотрим реализацию этих расширений, а также методы и типы межпроцессорных коммуникаций, особенности организации оперативной памяти для высокопроизводительных вычислительных систем.

Наиболее перспективным классом высокопроизводительных систем являются многопроцессорные системы. В организации многопроцессорных вычислительных систем следует выделить следующие ключевые моменты:

- количество и архитектура индивидуальных процессоров;
- структура и организация доступа к оперативной памяти;
- топология коммуникационной сети и ее быстродействие;
- работа с устройствами ввода/вывода.

Важнейшей характеристикой многопроцессорной вычислительной системы является ее *масштабируемость.* Масштабируемость является мерой, которая показывает, можно ли данную проблему решить быстрее, увеличив количество процессорных элементов. Данным свойством обладает как аппаратное, так и программное обеспечение.

Процессоры

Процессор является основным элементом вычислительной системы, определяющим

уровень его быстродействия. Повышение производительности вычислительной техники связано, в очень большой степени, с разработкой новых технологий и привлечением самых оригинальных и неожиданных решений устройства процессоров.

Конвейеры

Идея конвейера состоит в том, чтобы сложную операцию разбить на несколько более простых, таких, которые могут выполняться одновременно. Операция суммирования, например, включает вычитание порядков, выравнивание порядков, сложение мантисс и нормализацию. Каждая из подопераций может выполняться на отдельном блоке аппаратуры. При движении объектов по конвейеру одновременно на разных его участках (сегментах) выполняются различные подоперации, что дает увеличение производительности за счет использования параллелизма на уровне команд. При достижении объектом конца конвейера он окажется полностью обработанным. Конвейеры применяются как при обработке команд *(конвейеры команд),* так и в арифметических операциях *(конвейеры данных).* Поскольку использование конвейерной обработки усложняет конструкцию процессора, эффективным конвейер данных может быть при выполнении векторных операций. Операция над одним элементом данных в конвейере будет выполняться дольше, чем в обычном АЛУ.

Для эффективной реализации конвейера должны выполняться следующие условия:

- система выполняет повторяющуюся операцию;
- операция может быть разделена на независимые части;
- трудоемкость подопераций примерно одинакова.

Количество сегментов называют *глубиной конвейера.* Важным условием нормальной работы конвейера является отсутствие конфликтов, приводящих к простоям конвейера. Для этого объекты, поступающие в конвейер, должны быть независимыми. Если, например, операндом является результат предыдущей операции, возникают периоды работы конвейера ("конвейерные пузыри"), когда он пуст. "Пузырь" проходит по конвейеру, занимая место, но не выполняя при этом никакой полезной работы.

Теоретически, максимально достижимое увеличение быстродействия, которое можно получить с помощью конвейера данных, определяется формулой:

$$S = \frac{n \times d}{n + d},$$

где *n* — количество операндов, загружаемых в конвейер, *d* — глубина конвейера. Пусть требуется выполнить операцию сложения над двумя одномерными массивами по 200 элементов, причем операция суммирования состоит из пяти подопераций.

В этом случае ускорение составит

$$\frac{200 \times 5}{200 + 5} \approx 4{,}88.$$

Разумеется, это идеальная ситуация, недостижимая в реальной жизни. В частности, считается, что нет "конвейерных пузырей" и т. д.

Цикл выполнения команды состоит из нескольких шагов:

1. Выборка команды.

2. Декодирование команды, вычисление адреса операнда и его выборка.

3. Выполнение команды.

4. Обращение к памяти.

5. Запись результата в память.

Эти шаги могут выполняться на устройствах, организованных в конвейер (рис. 1.10), тогда обработка следующей команды может начинаться, пока идет обработка предыдущей команды. Для выполнения каждого этапа команды отводится один такт и в каждом новом такте начинается выполнение новой команды. Для хранения промежуточных результатов каждого этапа используется быстрая память (регистры). В результате, в каждом такте будут выполняться несколько (пять) команд и, несмотря на то, что время выполнения отдельной команды несколько увеличится, производительность системы в целом возрастет.

Рис. 1.10. *Конвейер команд*

Пусть имеется конвейер, состоящий из двух сегментов. Первый сегмент -выборка команды, второй — ее выполнение. Пусть время выполнения одной команды *T*\, тогда время выполнения 7 команд составит

$$\left(\frac{T_1}{2}\right) \times 8 = 4T_1.$$

Ускорение составляет 4.

В конвейере, состоящем из шести сегментов, время выполнения 7 команд составит уже

$$\left(\frac{T_1}{6}\right) \times 12 = 2T_1.$$

Для того чтобы задействовать все *N* сегментов конвейера, требуется *N* — 1 такт. После этого достигается максимальная производительность.

Казалось бы, увеличение числа сегментов будет увеличивать и скорость выполнения команд, однако с ростом глубины конвейера увеличиваются затраты времени на передачу информации между сегментами и синхронизацию, возрастает сложность аппаратной части, труднее избежать простоев конвейера.

Простой конвейера команд вызывается ситуацией, когда очередную команду из потока команд нельзя загрузить в конвейер сразу, а по какой-либо причине приходится ожидать несколько тактов. Если очередная команда в соответствующем ей такте не может быть выполнена, говорят о *конфликте.* В этом случае команда ожидает своей очереди, а пропускная способность конвейера падает. Ожидать своей очереди приходится и всем последующим командам. Команды, уже загруженные в конвейер, продолжают выполняться.

Принято выделять три типа конфликтов:

- структурные конфликты;
- конфликты по данным;
- конфликты по управлению.

Причиной *структурных конфликтов* является одновременный запрос на использование одного ресурса несколькими командами. Так, если одна команда обращается к оперативной памяти, в это же время другие команды не могут получить доступ к оперативной памяти. Другой пример структурного конфликта — недостаточное количество регистров для записи данных в одном такте. В этом случае работа конвейера приостанавливается до тех пор, пока запись вновь не окажется возможной. Избежать частого повторения структурных конфликтов можно, дублируя некоторые ресурсы, используя разделение кэш-памяти для данных и команд и т. д. Однако это усложняет конструкцию процессора и значительно повышает его стоимость.

Конфликты по данным являются следствием логической зависимости команд между собой и происходят, если для выполнения очередной команды требуется результат

выполнения предыдущей команды (ее результат выступает в качестве операнда). В этом случае команда не будет выполняться до тех пор, пока предыдущая команда не завершит свое выполнение и не передаст ей свой результат. Влияние таких зависимостей на эффективность работы конвейера определяется особенностями программ и архитектурой процессера. С помощью специальных методов можно снизить потери производительности в ситуациях такого рода, хотя избежать их полностью нельзя.

Конфликты по управлению вызываются наличием в программах условных конструкций. Выполняемая ветвь условного оператора определяется только после вычисления условия ветвления. Если учесть, что в программах до трети операторов являются ветвлениями, потери производительности вследствие простоев по управлению могут быть большими. В большинстве современных процессоров используются специальные устройства, которые выполняют выборку команд и помещают их в специальные очереди еще до того, как они могут понадобиться. Функциональные узлы конвейера умеют распознавать команды безусловного перехода, определяют адреса следующих команд и выбирают эти команды из очереди, что позволяет обеспечить непрерывный поток команд в конвейере. Условные переходы труднее поддаются оптимизации, но и здесь имеются способы повышения производительности конвейера при обработке условных переходов.

Избежать остановок конвейера (по крайней мере, частично) можно также, реорганизовав последовательность команд на этапе трансляции программы. Этот метод называется *планированием загрузки конвейера.*

Суперскалярные процессоры

Показателем эффективности работы конвейера является среднее количество тактов на выполнение команды (CPI — Cycles Per Instruction). Чем меньше эта величина, тем выше производительность процессора. Идеальной величиной является 1, однако значение CPI может быть и меньше единицы, если в одном такте параллельно выполняются несколько команд. Параллельное выполнение команд реализовано в *суперскалярных процессорах* и *процессорах со сверхдлинным командным словом.*

Это позволяет делать и конвейер, но при этом команды должны находиться на различных стадиях обработки (в разных сегментах конвейера). Суперскалярный процессор не только включает возможность конвейерной обработки, но и позволяет одновременно выполнять несколько команд в одном сегменте конвейера. Несколько команд одновременно могут выполняться в течение одного такта.

Суперскалярные процессоры используют параллелизм на уровне команд путем дублирования функциональных устройств и передачи в них нескольких команд из общего потока. Прежде всего, используют несколько конвейеров, работающих параллельно. Правда, параллельное выполнение команд в суперскалярных

процессорах не всегда возможно. Это может быть следствием любой из трех причин.

- *Конфликты по доступу к ресурсам.* Возникают, если несколько команд одновременно обращаются к одному ресурсу. Это может быть регистр, оперативная память или что-нибудь еще. Эта ситуация аналогична структурным конфликтам. Снизить отрицательный эффект подобных ситуаций можно дублированием устройств.
- *Зависимость по управлению.* У зависимости по управлению есть два аспекта. Первый — это проблемы, связанные с обработкой ветвлений. Второй связан с использованием команд переменной длины. В этом случае выборку следующей команды нельзя сделать, пока не завершено декодирование предыдущей команды. Таким образом, суперскалярная архитектура более всего подходит для RISC-процессоров с их фиксированным форматом и длиной команд.
- *Конфликты по данным.* Причиной конфликтов по данным являются зависимости по данным между командами, когда очередная операция не может быть выполнена, если ей требуется результат предыдущей операции. Такая зависимость является свойством программы и не может быть исключена компилятором или с помощью каких-то аппаратных решений. Избежать простоев можно, загрузив узлы выполнением других команд, пока формируется результат предыдущей операции.

Для разрешения возможных конфликтов используют методы внеочередной выборки и завершения команд, прогнозирование переходов, условное выполнение команд и др.

В суперскалярных процессорах используется динамическое распределение команд, причем порядок их выборки может не совпадать с порядком следования в программе, но при этом, разумеется, результат выполнения должен совпадать с результатом строго последовательного выполнения. Для эффективной реализации данного подхода последовательность команд, из которой производится выборка, должна быть достаточно большой — требуется довольно большое *окно выполнения* (Window of Execution).

Окно выполнения — это набор команд, которые являются кандидатами на выполнение в данный момент. Любая команда из этого окна может быть взята для исполнения с учетом вышеупомянутых ограничений. Количество команд в окне должно быть максимально большим.

Основными компонентами суперскалярного процессора являются устройства для интерпретации команд, снабженные логикой, позволяющей определить, являются ли команды независимыми, и достаточное число исполняющих устройств. В исполняющих устройствах могут быть конвейеры.

В суперскалярном процессоре в одном такте может выполняться обработка до

восьми команд (например, в процессорах Pentium — две, а в процессорах UltraSPARC — четыре). Это значение изменяется в процессе работы, поскольку практически неизбежные конфликты будут приводить к простоям оборудования. В результате этого производительность суперскалярного процессора оказывается переменной.

Почти все современные микропроцессоры, включая Pentium, PowerPC, Alpha и SPARC — суперскалярные. Примером компьютера с суперскалярным процессором является IBM RISC/6000. При тактовой частоте 62,5 МГц быстродействие системы на вычислительных тестах достигало 104 Мфлоп/с. (Мфлоп/с - "мегафлоп в секунду" единица измерения быстродействия процессора, составляющая один миллион операций с плавающей точкой в секунду). Суперскалярный процессор не требует специальных векторизующих компиляторов, хотя компилятор должен в этом случае учитывать особенности архитектуры.

Повышение производительности за счет увеличения числа функциональных устройств и использования низкоуровневого параллелизма можно эффективно реализовать, лишь обеспечив их сбалансированную загрузку. Здесь возможны два подхода. Первый — динамическая загрузка, основанная на анализе программного кода во время его выполнения. Второй подход — статический, поддерживаемый компилятором. Первый подход используется в суперскалярных процессорах, а второй — в процессорах со сверхдлинным командным словом.

Процессоры с сокращенным набором команд (RISC)

В основе RISC-архитектуры (RISC — Reduced Instruction Set Computer) процессора лежит идея увеличения скорости его работы за счет упрощения набора команд. Противоположную тенденцию представляют CISC-архитектуры, процессоры со сложным набором команд (CISC — Complete Instruction Set Computer). Основоположником архитектуры CISC является компания IBM, а в настоящее время лидером в данной области является Intel (процессоры Pentium). Идеи RISC-архитектуры использовались еще в компьютерах CDC6600 (разработчики — Крей, Торнтон и др.). Оба варианта относятся к противоположным границам *семантического разрыва* — увеличивающегося разрыва между программированием на языках высокого уровня и программированием на уровне машинных команд. В рамках CISC-подхода набор команд включает команды, близкие к операторам языка высокого уровня. В рамках RISC-подхода набор команд упрощается и оптимизируется под реальные потребности пользовательских программ.

Исследования показали, что 33% команд типичной программы составляют пересылки данных, 20% — условные ветвления и еще 16% — арифметические и логические операции. В подавляющем большинстве команд вычисление адреса может быть выполнено быстро, за один цикл. Более сложные режимы адресации используются примерно в 18% случаев. Около 75% операндов являются скалярными, т. е. переменными целого, вещественного, символьного типа и т. д., а остальные являются

массивами и структурами. 80% скалярных переменных — локальные, а 90% структурных являются глобальными. Таким образом, большинство операндов — это локальные операнды скалярных типов. Они могут храниться в регистрах.

Согласно статистике, большая часть времени тратится на обработку операторов CALL (вызов подпрограммы) и RETURN (возврат из подпрограммы). При компиляции эти операторы порождают длинные последовательности машинных команд с большим числом обращений к памяти, поэтому даже если доля этих операторов составляет всего 15%, они потребляют основную часть процессорного времени. Только около 1% подпрограмм имеют более шести параметров, а около 7% подпрограмм содержат более шести локальных переменных.

В результате изучения этой статистики был сделан вывод о том, что в типичной программе доминируют простые операции: арифметические, логические и пересылки данных. Доминируют и простые режимы адресации. Большая часть операндов — это скалярные локальные переменные. Одним из важнейших ресурсов повышения производительности является оптимизация операторов CALL и RETURN.

В основу RISC-архитектуры положены следующие принципы и идеи. Набор команд должен быть ограниченным и включать только простые команды, время выполнения которых после выборки и декодирования один такт или чуть больше. Используется конвейерная обработка. Простые RISC-команды допускают эффективную аппаратную реализацию, в то время как сложные команды CISC могут быть реализованы только средствами микропрограммирования. Конструкция устройства управления в случае RISC-архитектуры упрощается, и это дает возможность процессору работать на больших тактовых частотах. Использование простых команд позволяет эффективно реализовать и конвейерную обработку данных, и выполнение команд.

Произведем подсчет. Пусть 80% команд — программы простые и 20% — сложные. Пусть также простые команды процессор с CISC-архитектурой выполняет за 4 такта, сложные — за 8 тактов, а длительность одного такта 100 наносекунд. Эти цифры близки к реальным. Простую команду RISC-процессор выполняет за 1 такт, а сложную — за 14 тактов, длительность такта 75 наносекунд. На выполнение программы, содержащей 1 000 000 команд,

CISC-процессор затратит 10^6x0,80x4x 10^{-7} с + 10^6x0,20x8x 10^{-7} с = 0,48 с, а RISC-процессор - 10^6 x 0,80 x 1 x 10^{-7} с + 10^6 x 0,20 x 14 x 10^{-7} с = 0,27 с, т. е. почти вдвое меньше.

Сложные команды RISC-процессором выполняются дольше, но их количество относительно невелико. Простые команды CISC-процессором выполняются не очень быстро, что объясняется сложностью реализации команд в данной архитектуре. В RISC-процессорах небольшое число команд адресуется к памяти. Выборка данных из оперативной памяти требует более одного такта. Большая часть команд работает с

операндами, находящимися в регистрах. Все команды имеют унифицированный формат и фиксированную длину. Это упрощает и ускоряет загрузку и декодирование команд, поскольку, например, код операции и поле адреса всегда находятся в одной и той же позиции. Переменные и промежуточные результаты вычислений могут храниться в регистрах. С учетом статистики использования переменных, большую часть локальных переменных и параметров процедур можно разместить в регистрах. При вызове новой процедуры содержимое регистров обычно перемещается в оперативную память, однако, если количество регистров достаточно велико, удается избежать значительной части длительных операций обмена с памятью, заменив их операциями с регистрами. Благодаря упрощенной архитектуре RISC-процессора, на микросхеме появляется место для размещения дополнительного набора регистров. Распределение регистровой памяти под переменные выполняется компилятором.

Несмотря на упомянутые преимущества RISC-архитектуры, нет простого и однозначного ответа на вопрос о том, что лучше — RISC или CISC. Так, например, для RISC-архитектуры характерны повышенные требования к оперативной памяти. Примером CISC-процессора является процессор Pentium (количество команд более 200, длина команды 1—11 разрядов, имеется 8 регистров общего назначения), а в качестве примера RISC-процессора можно привести SunSPARC (количество команд около 50, длина команды 4 разряда, имеется 520 регистров общего назначения).

В настоящее время вычислительные системы с RISC-архитектурой занимают лидирующие позиции на мировом компьютерном рынке рабочих станций и серверов. Развитие RISC-архитектуры связано с развитием компиляторов, которые должны эффективно использовать преимущества большого регистрового файла, конвейеризации и т. д.

Процессоры со сверхдлинным командным словом

В суперскалярных процессорах вопрос о параллелизме команд решается ап-паратно, аппаратно реализованы и методы снижения потерь быстродействия от разного рода зависимостей. Между суперскалярными процессорами имеется совместимость программ на уровне исполняемых (бинарных) файлов, добавление новых параллельных функциональных узлов может повлиять на выполнение программы только в сторону увеличения скорости ее выполнения. Но у них есть и свои недостатки. Это, прежде всего, сложность аппаратной части, а также ограниченный размер окна выполнения, что уменьшает возможности определения потенциально параллельных команд.

Альтернативой суперскалярным процессорам являются процессоры со сверхдлинным командным словом (VLIW-- Very Large Instruction Word). Работа VLIW-процессора основана на выявлении параллелизма команд во время трансляции. Транслятор анализирует программу, определяя, какие операции могут выполняться параллельно. Такие операции "упаковываются" в одну большую команду (рис. 1.11). После того, как

"большая" команда выбрана из памяти, составляющие ее обычные команды выполняются параллельно. В этом случае решается проблема окна выполнения, поскольку транслятор анализирует всю программу в целом в поисках параллельных операций.

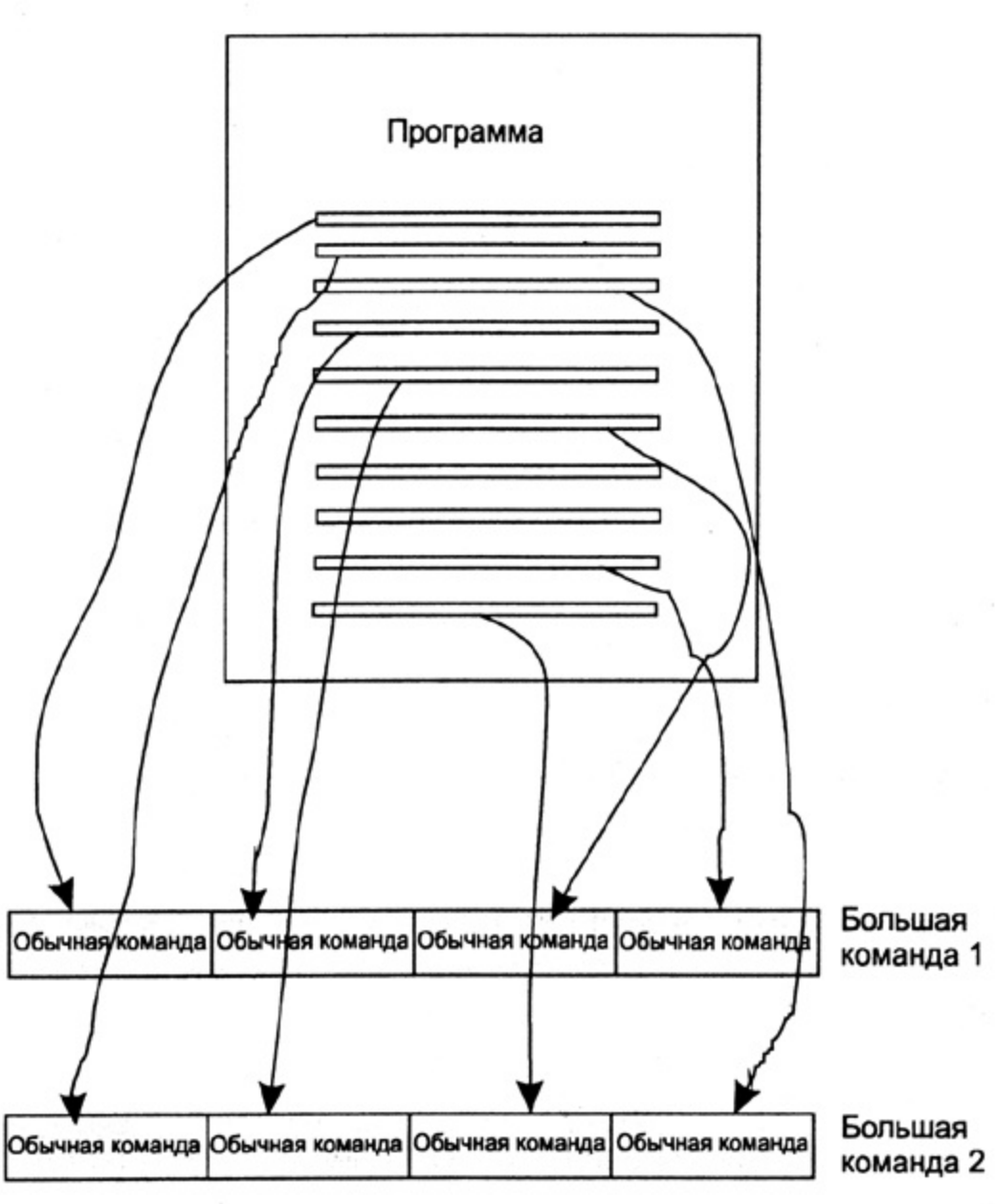


Рис. 1.11. *Создание последовательности VLIW-команд*

Схема VLIW-процессора приведена на рис. 1.12.

В VLIW-процессорах количество функциональных узлов можно увеличивать, не усложняя остальную аппаратную часть, что приходится делать в RISC-процессорах.

Вместе с тем, есть и свои проблемы. Для хранения данных и обслуживания многочисленных функциональных узлов требуется большое количество регистров. Между функциональными устройствами и регистрами, между регистрами и оперативной памятью возникает интенсивный обмен данными. Кэш-память команд и устройство выборки должен связывать канал с большой разрядностью — если

размер обычной команды 24 бита, а одно сверх-

большое слово содержит 7 простых команд, в результате получим 168 битов на одну команду. Увеличивается размер исполняемого кода, теряется совместимость на уровне бинарных файлов.

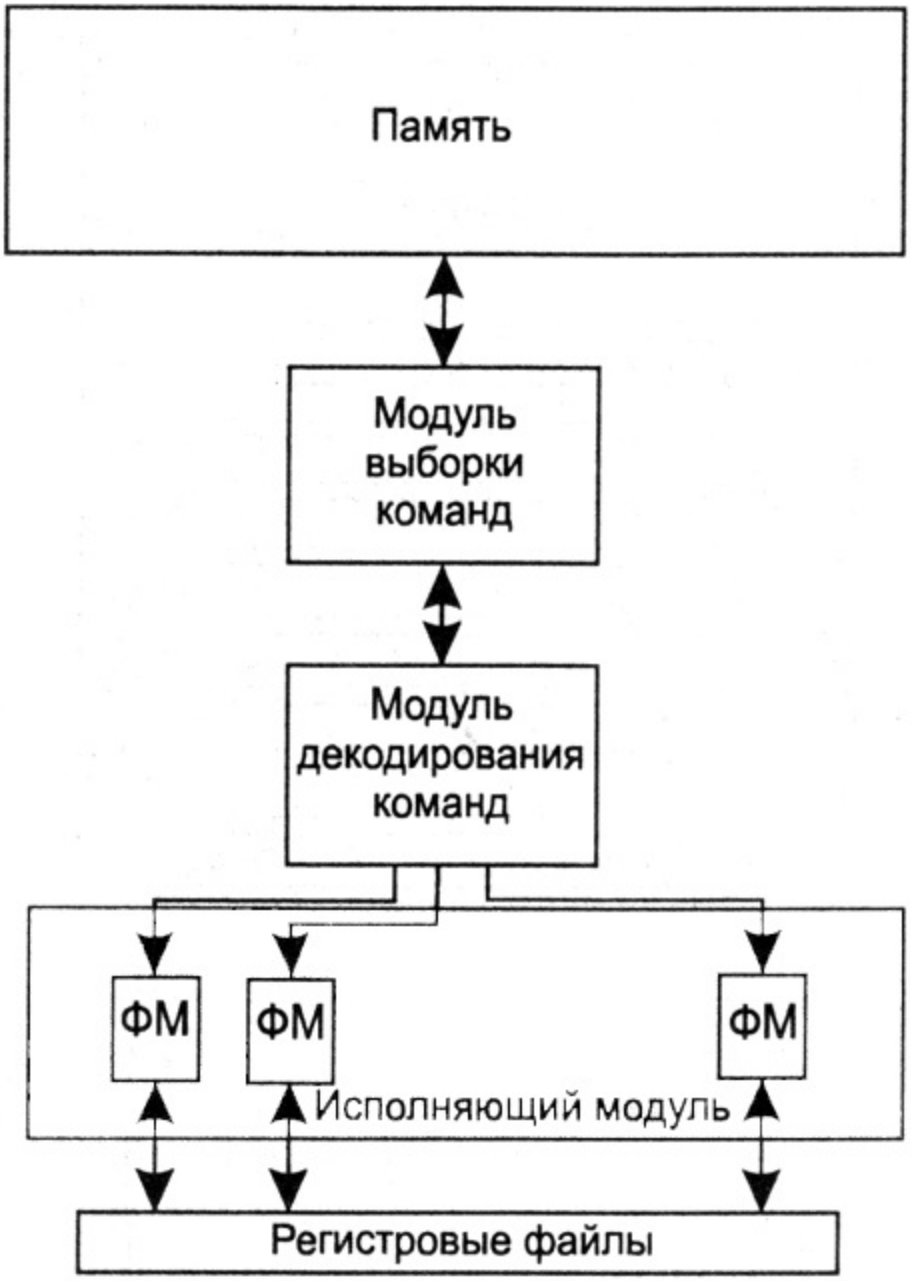


Рис. 1.12. *Схема процессора со сверхдлинным командным словом*

Эффективность использования данной архитектуры зависит от качества трансляторов.

В трансляторах для VLIW-процессоров применяются такие специальные приемы, как развертка циклов, предсказание ветвлений и планирование выдачи команд. Метод трассировочного планирования заключается в том, что из последовательности обычных команд исходной программы длинные команды генерируются путем

просмотра (трассировки) линейных (без ветвлений) участков программы.

В отличие от традиционной архитектуры, регистры не резервируются для определенных целей, их использование определяется во время компиляции. Высокая скорость работы оперативной памяти достигается ее расслоением. Примерами процессоров с VLIW-архитектурой являются Itanium, а также некоторые процессоры, ориентированные на мультимедиа-приложения.

Векторная обработка данных

В набор команд векторного процессора входят не только скалярные команды, но и команды, операндами которых являются массивы (векторы), по-

этому векторный процессор может обработать одной командой сразу множество значений. Пусть A_1, A_2 и p — это три массива, имеющие одинаковую размерность и одинаковую длину, и имеется оператор:

$$P = A_1 + A_2.$$

Векторный процессор за один цикл выполнения команды произведет поэлементное сложение массивов A_1 и A_2 и присвоит полученные значения соответствующим элементам массива p. Каждый операнд при этом хранится в особом, векторном регистре. Обычному, последовательному процессору пришлось бы многократно исполнить, операцию сложения элементов двух массивов. Векторный процессор выполняет лишь одну команду. Разумеется, реализация такой команды будет более сложной.

Векторные операции могут быть реализованы с использованием параллелизма, который обеспечивают конвейеризованные функциональные узлы. Такие архитектуры и называются векторными процессорами. Можно считать, что векторный процессор построен на основе SISD-архитектуры, в которой реализованы векторные операции. В результате этого получаем SIMD-архитектуру. В векторных процессорах используются векторные регистры объемом 64 или 128 машинных слов.

Векторные команды:

- загружают операнд из оперативной памяти в векторный регистр;
- записывают операнд из регистра в память;
- выполняют арифметические и логические операции над векторами;
- выполняют другие операции над векторами и скалярами.

С точки зрения программиста, при работе на компьютере с векторной архитектурой в программе можно использовать обычные операции над массивами, которые транслятор переводит в векторные машинные команды.

Векторный модуль содержит несколько векторных регистров общего назначения, регистр длины вектора (для хранения длины обрабатываемого вектора), регистр маски. Регистр маски содержит последовательность битов — двоичные разряды вектора, которым соответствуют нулевые биты маски, в определенных ситуациях игнорируются при выполнении векторных операций.

Очевидно, за счет векторизации можно надеяться получить высокую производительность. Мы не случайно делаем здесь эту оговорку, т. к. простой перенос обычной программы на векторный компьютер не гарантирует увеличение быстродействия. Кроме того, векторным ЭВМ присущи и другие особенности. Количество машинных команд, необходимых для выполнения одной и той же программы, использующей операции с массивами, меньше в случае векторного процессора, чем обычного, скалярного. Уменьшение потока команд позволяет снизить требования к устройствам коммуникации, в том числе к каналам связи между процессором и оперативной памятью компьютера. При соответствующей организации оперативной памяти данные в процессор будут передаваться на каждом такте, что дает значительный выигрыш в производительности компьютера.

Следует заметить, что именно векторные ЭВМ были первыми высокопроизводительными компьютерами (векторный компьютер Сеймора Крэя) и, традиционно, именно ЭВМ с векторной архитектурой назывались суперкомпьютерами.

Векторные компьютеры различаются тем, как операнды передаются командам процессора. Здесь можно выделить следующие основные схемы:

- *из памяти в память* — в этом случае операнды извлекаются из оперативной памяти, загружаются в АЛУ и результат возвращается в оперативную память;
- *из регистра в регистр* — операнды сначала загружаются в векторные регистры, затем операнд передается в АЛУ и результат возвращается в один из векторных регистров.

Преимущество первой схемы заключается в том, что она дает возможность работать с векторами произвольной длины, тогда как во втором случае требуется разбиение длинных векторов на части, длина которых соответствует возможностям векторного регистра. С другой стороны, в первом случае имеется определенное *время запуска,* которое должно пройти между инициализацией команды и появлением в конвейере первого результата. Если конвейер уже загружен, результат на его выходе будет появляться в каждом такте. Примером ЭВМ с такой архитектурой являются компьютеры серии CYBER 200, время запуска у которых составляло до 100 тактов. Это очень большая величина, даже при работе с векторами длиной 100 элементов, вышеупомянутые компьютеры достигали лишь половины от своей максимально возможной производительности.

В векторных компьютерах, работающих по схеме регистр-регистр, длина вектора

гораздо меньше. Для компьютеров серии Cray эта длина составляет всего 64 слова, однако, существенно меньшее время запуска позволяет увеличить быстродействие. Правда, если работать с длинными векторами, их приходится разбивать на части меньшей длины, что снижает быстродействие. Векторные компьютеры, работающие по схеме из регистра в регистр, в настоящее время доминируют на рынке векторных компьютеров и наиболее известными представителями этого семейства являются компьютеры фирм Cray, NEC, Fujitsu и Hitachi.

Архитектура типичного векторного суперкомпьютера Cray следующая (в качестве примера рассмотрим модель Cray-1). Процессор имеет 8 векторных регистров, каждый из которых может хранить 64 слова по 64 бита каждое. Есть также 8 скалярных регистров для 64-битовых слов и 8 регистров адресации для 20-битовых слов. Вместо кэш-памяти используются специальные регистры, управление которыми осуществляется программным путем. Всего компьютер Cray-1 содержит до 12 конвейеризованных процессоров, имеющих отдельные конвейеры для различных арифметических и логических операций. Скорость работы процессора строго согласована со скоростью работы оперативной памяти.

Мультимедийные приложения обычно работают с большими массивами данных, состоящими из коротких (8- или 16-разрядных) значений с фиксированной точкой. Такие приложения представляют огромный потенциал векторного (SIMD) параллелизма, поэтому новые поколения микропроцессоров общего назначения снабжаются мультимедийными командами. Мультимедийные расширения включены в системы команд процессоров Intel (MMX-расширение системы команд Pentium и новые SIMD-команды Pentium III), AMD (3D Now!), Sun (VIS SPARC), Compaq (Alpha MVI), Hewlett Packard (PA-RISC MAX2), SGI (MDMX), Motorola (PowerPC AltiVec).

В Pentium III введена параллельная обработка в режиме SIMD-процессора четырех 32-разрядных операндов в формате с плавающей точкой. Реализация векторного параллелизма в мультимедийных расширениях имеет свои особенности. Это, прежде всего, операции над подсловами. При длине слова, например, 64 бита и длине операнда 16 битов, можно одновременно выполнять действия с 4 операндами.

Процессоры для параллельных компьютеров

Принято выделять четыре уровня параллелизма:

1. *Параллелизм заданий* — каждый процессор загружается своей собственной, независимой от других, вычислительной задачей. Параллелизм такого типа представляет интерес скорее для системных администраторов, чем рядовых пользователей.

2. *Параллелизм на уровне программы* — вычислительная программа разбивается на части, которые могут выполняться одновременно на различных процессорах.

3. *Параллелизм команд* — обычно реализован на низком уровне, это, например, конвейеры и т. д.

4. *Параллелизм на уровне машинных слов и арифметических операций* —- в некоторых ситуациях, например, сложение двух операндов выполняется одновременным сложением всех их двоичных разрядов.

Как видим, в первом и втором случаях нет необходимости в какой-то особой архитектуре процессора, и лишь в третьем и четвертом случаях требуется введение новых элементов, речь о которых шла выше. Но все они не являются специфическими для параллельной многопроцессорной архитектуры.

Если в 80—90-е годы прошлого века в высокопроизводительных вычислительных системах доминировали векторные процессоры, то сейчас они уступили место лидера RISC-процессорам. Современный RISC-процессор обычно уступает по тактовой частоте процессорам Pentium III и IV, однако его производительность на операциях с плавающей точкой выше.

Процессоры Pentium

Процессоры Pentium не применяются в интегрированных параллельных системах, но являются самым распространенным видом процессоров в кластерах рабочих станций. В современных поколениях процессоров Pentium быстродействие достигается как за счет увеличения тактовой частоты, так и благодаря применению глубокой конвейеризации операций. В процессорах Pentium IV конвейеры содержат не менее 20 сегментов. Применяется также специальная организация кэш-памяти (маленький объем кэш-памяти первого уровня, всего 8 Кбайт, позволяет сократить цикл выборки данных из памяти), планирование выполнения команд, особая конструкция шины и другие методы.

Процессор Compaq Alpha EV67/68

В момент выпуска на рынок тактовая частота этого процессора составляла 833 МГц. В каждом такте выдаются два результата операций с плавающей точкой.

Имеются 2 целочисленных *регистровых файла* с 80 входами каждый и один — с плавающей точкой (72 входа). Регистровый файл — это набор регистров центрального процессора. Каждый целочисленный регистровый файл обслуживает свой блок функциональных узлов целочисленной арифметики, которым присвоены наименования "кластер 0" и "кластер 1".

Целочисленное умножение полностью конвейеризовано. Два целочисленных "кластера" и два модуля арифметики с плавающей точкой позволяют одновременно" обрабатывать *6* команд.

Имеются кэш-память команд и кэш-память данных (обе по 64 Кбайт). Поддерживаются методы динамического планирования выполнения программ, такие как выполнение по предположению, предсказание ветвлений и др. Время задержки для пересылки целых значений в память составляет 3 цикла, а значений с плавающей точкой 4 цикла. Схема процессора приведена на рис. 1.13.

Многопоточная архитектура

В настоящее время появились и развиваются процессоры с многопоточной архитектурой (MTA — MultiThread Architecture). Идея, положенная в их основу, заключается в том, что программа выполняется в несколько потоков и

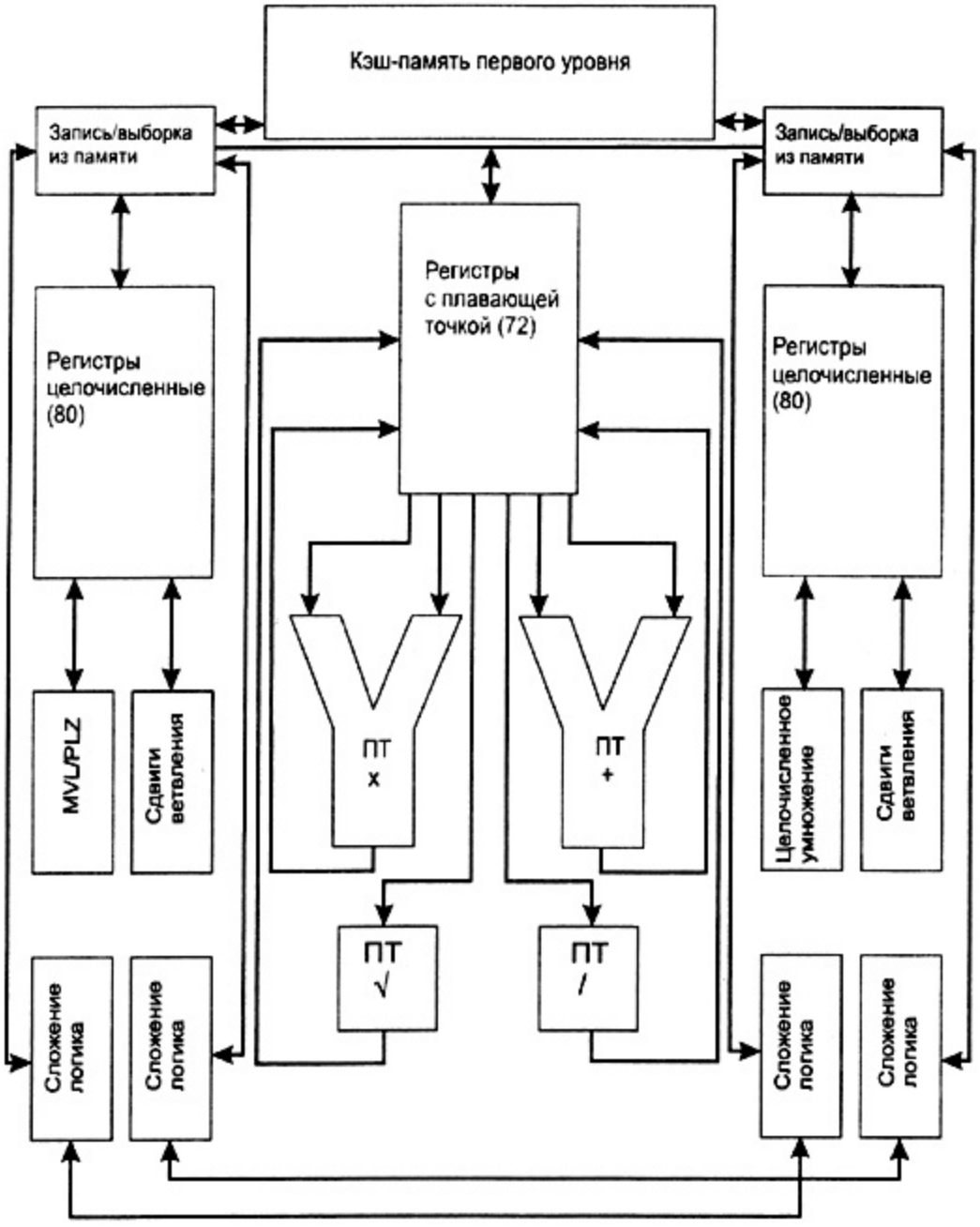


Рис. 1.13. *Схема процессора Alpha EV67/68 (ПТ — модуль операций с плавающей точкой)*

каждому потоку отдается специальный регистровый файл. Задача создания и управления множеством потоков решается транслятором. Пример компьютера с многопоточной архитектурой — Cray MTA-2. Разновидностью данного подхода является одновременная многопоточность (SMT — Simultaneous MultiThreading), когда один физический процессор видится операционной системе как два "логических" процессора.

Оперативная память

Характеристики оперативной памяти и особенности ее устройства являются важнейшим фактором, от которого зависит быстродействие компьютера. Ведь даже при наличии быстрого процессора скорость выборки данных из памяти может оказаться невысокой, и именно эта невысокая скорость работы с оперативной памятью будет определять быстродействие компьютера. Время цикла работы с памятью t_m обычно заметно больше, чем время цикла центрального процессора t_c. Если процессор инициализирует обращение к памяти, она будет занята в течение времени $t_c + t_m$ и в течение этого промежутка времени ни одно другое устройство, в том числе и сам процессор, не смогут работать с оперативной памятью. Таким образом, возникает проблема доступа к памяти.

Эта проблема решается специальной организацией оперативной памяти. Принята следующая классификация параллельных компьютеров по архитектуре подсистем оперативной памяти:

- *системы с разделяемой памятью,* у которых есть одна большая виртуальная память, и все процессоры имеют одинаковый доступ к данным и командам, хранящимся в этой памяти;
- *системы с распределенной памятью,* у которых каждый процессор имеет свою локальную оперативную память, и к этой памяти у других процессоров нет доступа.

Различие между разделяемой и распределенной памятью заключается в способе интерпретации адреса. Это различие можно пояснить на следующем примере. Пусть один из процессоров выполняет команду load r1*, i* (загрузить в регистр r1 данные из ячейки памяти i). Если команда выполняется на компьютере с разделяемой памятью, номер ячейки i имеет одинаковый смысл для всех процессоров, т. е. i является глобальным адресом. В системе с распределенной памятью ячейка i разная для различных процессоров и в регистры r1 по этой команде будут загружены разные значения.

Для программиста часто бывает важно знать тип оперативной памяти компьютера, на котором он работает. Ведь архитектура памяти определяет способ взаимодействия между различными частями параллельной программы. При выполнении на компьютере

с распределенной памятью программа перемножения матриц, например, должна создать копии перемножаемых матриц на каждом процессоре, что технически осуществляется передачей на эти процессоры сообщения, содержащего необходимые данные. В случае системы с разделяемой памятью достаточно лишь один раз задать соответствующую структуру данных и разместить ее в оперативной памяти. Остановимся подробнее на различных архитектурах подсистемы памяти.

Разделяемая память

Простейший способ создать многопроцессорный вычислительный комплекс с разделяемой памятью — взять несколько процессоров, соединить их общей шиной между собой и с оперативной памятью. Это — простой, но не самый лучший способ, поскольку, если один процессор принимает команду или передает данные, все остальные процессоры вынуждены переходить в режим ожидания. Это приводит к тому, что, начиная с некоторого числа процессоров, быстродействие такой системы перестает увеличиваться при добавлении нового процессора.

Мы уже выяснили, что несколько улучшить картину может применение кэш-памяти для хранения команд. При наличии локальной, т. е. принадлежащей данному процессору, кэш-памяти, следующая необходимая ему команда с большой вероятностью будет находиться в кэш-памяти. В результате этого уменьшается количество обращений к шине и быстродействие системы возрастает.

Кэш-память действует следующим образом. Если центральный процессор обращается к оперативной памяти, вначале запрос на необходимые данные поступает в кэш-память. Если они там уже находятся, выполняется быстрая пересылка данных в регистр процессора, в противном случае данные считываются из оперативной памяти и помещаются в кэш-память, а из нее уже загружаются в регистр.

При использовании в многопроцессорных вычислительных системах возникает проблема *кэш-когерентности.* Она заключается в том, что если двум или более процессорам понадобилось значение одной и той же переменной, оно будет храниться в виде нескольких копий в кэш-памяти всех процессоров. Один из процессоров может изменить это значение в результате выполнения своей команды и оно будет передано в оперативную память компьютера. Но в кэш-памяти остальных процессоров все еще хранится старое значение. Следовательно, необходимо обеспечить своевременное обновление данных в кэш-памяти всех процессоров компьютера.

Проблема кэш-когерентности может решаться программным путем -- на уровне транслятора и операционной системы. Транслятор, например, может определять моменты безопасной синхронизации кэш-памяти при выполнении программы. Другой подход основан на аппаратном решении проблемы кэш-когерентности. В этом случае применяются специальные протоколы, кэш-память используется более эффективно, все происходит "прозрачно" для программиста. *Протоколы каталогов* реализуют

сбор и учет информации о копиях данных в кэш-памяти. Эта информация хранится в специальной области оперативной памяти — каталоге. Запросы проверяются с помощью каталога, затем выполняются необходимые пересылки данных. Использование данного подхода эффективно в больших системах со сложными схемами коммуникации. *Протоколы слежения* распределяют "ответственность" за когерентность кэшей между контроллерами кэш-памяти. Контроллер определяет изменение содержания кэш-памяти и передает информацию об этом другим модулям кэш-памяти. Очевидно, в этом случае возрастает объем передаваемых данных. В некоторых системах используется адаптивная смесь обоих подходов.

Имеются различные реализации разделяемой памяти. Это, например, разделяемая память с дискретными модулями памяти. Физическая память состоит из нескольких модулей, хотя виртуальное адресное пространство остается общим. Вместо общей шины в этом случае используется переключатель, направляющий запросы от процессора к памяти. Такой переключатель может одновременно обрабатывать несколько запросов к памяти, поэтому, если все процессоры обращаются к разным модулям памяти, быстродействие возрастает.

Вычислительной системой с разделяемой памятью является компьютер KSR-1 фирмы Kendall Square Research.

Чередуемая память

Чередуемая память разделяется на банки памяти. Принято соглашение 0 том, что ячейка памяти с номером i находится в банке памяти с номером 1 mod n, где n — количество банков памяти, а mod — операция вычисления остатка от деления. Таким образом, если имеется 8 банков памяти, то первому банку памяти будут принадлежать ячейки памяти с номерами 0, 8, 16, …, второму — 1, 9, 17, … и т. д. Запросы к различным банкам памяти могут обрабатываться одновременно. При достаточном количестве банков памяти скорость обмена данными между памятью и процессором может быть близка к идеальному значению — одно машинное слово за один такт работы процессора.

Ячейки памяти могут быть перенумерованы и непрерывным образом, т. е., скажем, в первом банке находятся ячейки с номерами от 0 до 255, во втором от 256 до 511 и т.д. В векторных компьютерах обычно используется первый способ адресации, а в многопроцессорных комплексах с разделяемой памятью — второй.

Распределенная память

В вычислительных системах с распределенной памятью оперативная память есть у каждого процессора. Процессор имеет доступ только к своей памяти. В этом случае отпадает необходимость в шине или переключателе, нет и конфликтов по доступу к памяти. Нет присущих системам с разделяемой памятью ограничений на число

процессоров, нет и проблемы с кэш-когерентностью. Но, с другой стороны, усложняется организация обмена данными между процессорами. Обычно такой обмен осуществляется при помощи обмена сообщениями-посылками, содержащими данные. Для фор-

мирования такой посылки требуется время, для получения и считывания полученных данных — тоже. Эти дополнительные затраты — плата за те преимущества, о которых шла речь.

Программирование для систем с распределенной памятью — более сложная задача. Оно требует разбиения исходной вычислительной задачи на подзадачи, выполнение которых может быть поручено разным процессорам.

Одним из компьютеров такого типа являлась вычислительная система CM-5 фирмы Thinking Machines. Она состояла из процессорных элементов, построенных на основе микропроцессора SPARC и соединенных сетью со специальной топологией типа "дерево". У каждого процессорного элемента имелась локальная память объемом 32 мегабайта.

В схеме с распределенно-разделяемой памятью каждый ПЭ имеет собственный модуль памяти, но адресное пространство — общее.

Связь между элементами параллельных вычислительных систем

Важнейшим элементом архитектуры любого компьютера, а высокопроизводительных вычислительных систем в особенности, является коммуникационная сеть, связывающая процессоры с оперативной памятью, процессоры между собой, процессоры с другими устройствами. Она оказывает решающее влияние на производительность системы. Трафик в такой сети состоит из пересылаемых данных и команд. Основной характеристикой сети является пропускная способность, измеряемая в битах в секунду.

Далее речь пойдет о многопроцессорных компьютерах, которые в данном контексте принято рассматривать как набор узлов (процессорных элементов, модулей памяти, переключателей) и соединений между узлами.

Приведем несколько определений.

- Организация внутренних коммуникаций вычислительной системы называется ее *топологией.*
- Два узла именуются *соседними,* если между ними имеется прямое соединение.
- *Порядком узла* называется количество его соседей.
- *Коммуникационным диаметром* сети именуется максимальный путь между любыми двумя узлами.

- *Масштабируемость* характеризует возрастание сложности соединений при добавлении в конфигурацию новых узлов. Если система обладает высокой степенью масштабируемости, ее сложность будет незначительно изменяться при наращивании системы, неизменным будет и диаметр сети.

Есть два типа топологий коммуникационных сетей — *статические* и *динамические.* В случае статической сети все соединения фиксированы, а в динамической сети в межпроцессорных соединениях используются переключатели. Первый тип сетей более всего подходит для решения тех задач, в которых известны структура и характер обмена данными. Динамические соединения более универсальны, но они дороже, их реализация сложнее.

Статические топологии

Соединение с помощью одиночной шины является самым простым и дешевым (рис. 1.14). Основной его недостаток заключается в том, что в каждый момент времени возможна только одна пересылка данных/команд. Пропускная способность обратно пропорциональна количеству процессоров, подключенных к шине. Такой способ соединения хорош только для систем, содержащих не более 10 процессоров.

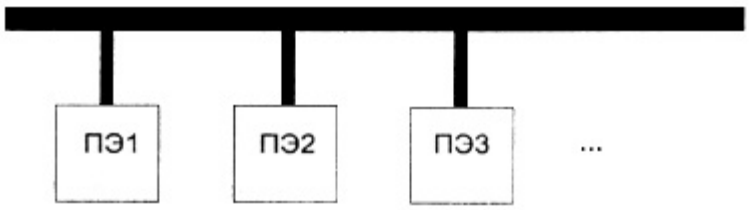
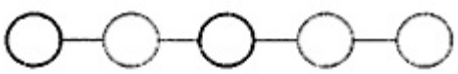


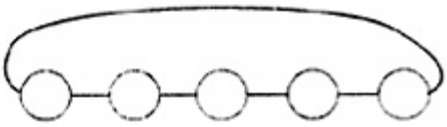
Рис. 1.14. *Соединение с помощью шины*

Стандарт IEEE P896 для высокопроизводительной шины, который получил название Scalable Coherent Interface (SCI — масштабируемый когерентный интерфейс), позволяет отчасти решить проблему сравнительно низкой скорости работы шины, но данное устройство имеет более сложную организацию, чем простая шина. Такая шина применялась в системе HP/Convex SPP-2000.

Более эффективным является другой способ соединения — одномерная решетка. У каждого элемента в этом случае есть две связи с соседями, а граничные элементы имеют по одной связи (рис. 1.15). Если замкнуть концы одномерной решетки, получим топологию "кольцо".



а



б

Рис. 1.15. *Одномерная решетка (а) и кольцо (б)*

Обобщением одномерной решетки является ^-мерная решетка, в которой у каждого ПЭ имеется 2*d* связей. Особый случай представляют граничные элементы, число связей у которых может быть разным. Такой тип коммуникационной сети не подходит для создания многопроцессорных конфигураций с большим числом процессоров, поскольку максимальная задержка передачи сообщений от одного из N процессоров к другому пропорциональна корень из N и быстро растет с увеличением числа процессоров. Двумерная решетка представлена на рис. 1.16.

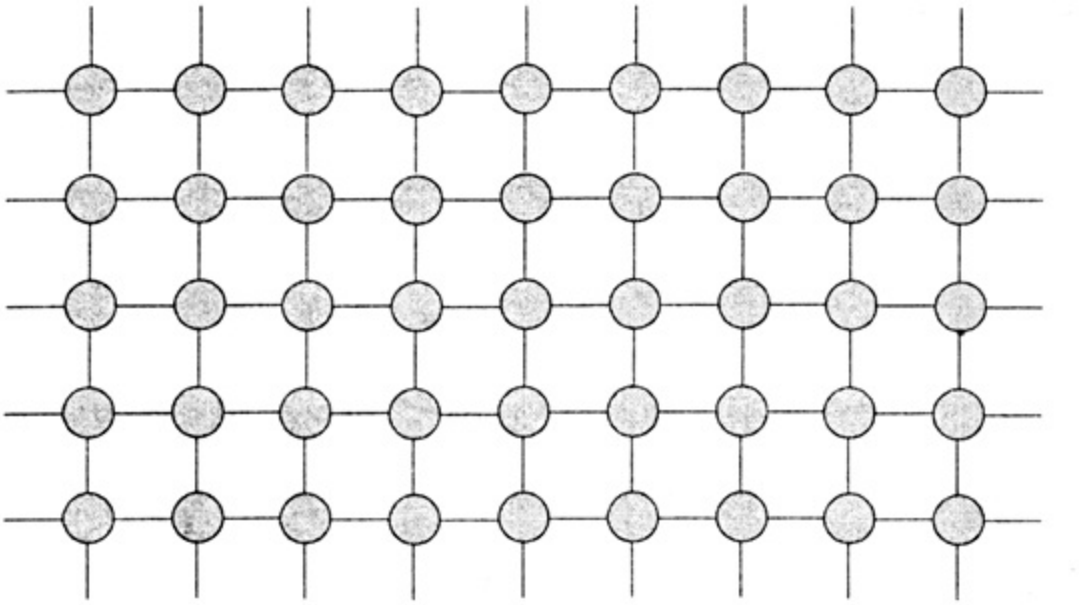


Рис. 1.16. *Плоская решетка*

Двумерная решетка обеспечивает хорошее быстродействие. Для передачи данных между процессорными элементами необходимо определить маршрут пересылки данных, при этом для решетки размером *п* х *п* требуется максимум 2 х (я — 1) промежуточных узлов. Достаточно часто используются и трехмерные решетки. Популярность двумерных и трехмерных решеток связана с тем, что многие задачи научно-технического характера имеют соответствующую структуру.

В топологии "звезда" есть один центральный узел, с которым соединяются все остальные процессорные элементы. Таким образом, у каждого ПЭ имеется одно соединение, а у центрального ПЭ — *N*— *I* соединение (рис. 1.17).

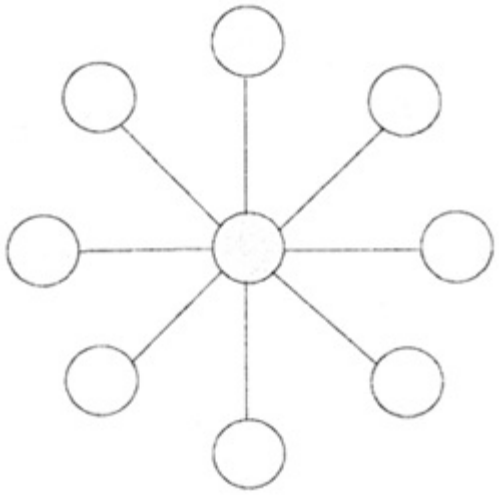


Рис. 1.17. *Соединение "звезда"*

В такой топологии слабым звеном оказывается центральный узел, поэтому она тоже не подходит для больших систем.

В сети с полносвязной топологией (рис. 1.18) все процессорные элементы связаны друг с другом. Пересылки могут выполняться одновременно между различными парами процессорных элементов. В такой системе легко могут быть реализованы широковещательные и многоадресные пересылки. Как быстродействие, так и стоимость системы высоки, причем стоимость существенно возрастает с увеличением числа процессорных элементов.

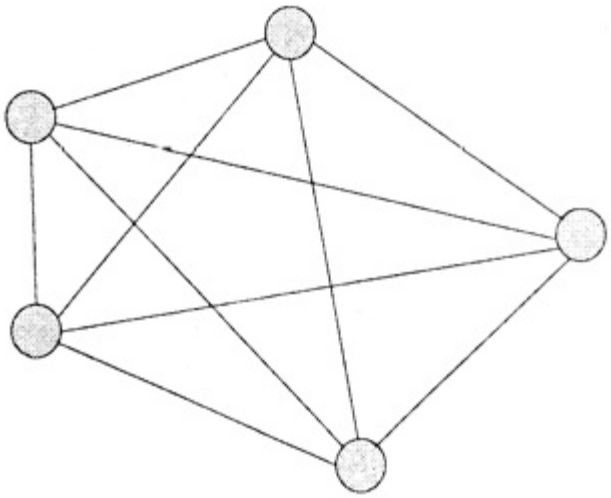


Рис. 1.18. *Сеть с полносвязной топологией*

Довольно распространенной является хорошо масштабируемая топология "гиперкуб" (рис. 1.19). В ней 2" процессорных элементов могут быть организованы в n-мерный гиперкуб. У каждого ПЭ имеется *п* соединений. Здесь также требуется маршрутизация пакетов с данными, причем максимальное количество промежуточных узлов равно *п.* Оно и определяет максимальную задержку передачи данных.

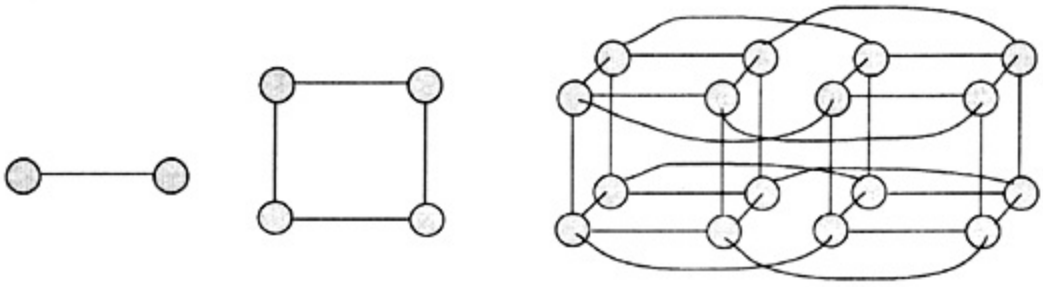


Рис. 1.19. *Одномерный гиперкуб (а), двумерный гиперкуб (б),* четырехмерный гиперкуб *(в)*

Для адресации узлов в гиперкубе каждому узлу присваивается свой идентификационный номер, при этом двоичные представления идентификационных номеров соседних узлов отличаются одним разрядом. Алгоритм пересылки сообщения от одного узла к другому в этом случае достаточ-

но простой и основан на побитовом сравнении двоичных представлений идентификационных номеров текущего узла и адресата. Например, в 4-мерном гиперкубе связаны ПЭ с номерами 1 (0001), 2 (0010), 4 (0100) и 8 (1000). Такая нумерация узлов называется *схемой кода Грея.*

Маршрутизация

В решеточных топологиях в каждый момент времени выполняется пересылка пакетов данных вдоль одного определенного измерения. В трехмерном случае, например, путь сообщения от узла *(a, b, c) к* узлу *(A, B, C)* будет пролегать сначала вдоль первого измерения к узлу *(a, B,* c), затем вдоль второго измерения к узлу *(A, B, c)* и, наконец, вдоль третьего измерения к узлу *(A, B, C)*.

Маршрутизация в топологиях "звезда" простая. Путь сообщения всегда проходит через центральный узел.

Гиперкуб представляет собой d-мерную решетку с двумя узлами по каждому измерению, поэтому маршрутизация выполняется почти так же, как и в решетках. Отличие состоит в том, что путь от узла *A* к узлу *B* вычисляется с помощью операции "исключающее ИЛИ" (XOR):

$$X = A \otimes B,$$

применяемой к двоичному представлению номеров узлов. Если i-и бит *X* равен 1, сообщение пересылается соседнему узлу в i-м измерении. Если же этот бит равен нулю, сообщение не пересылается.

Динамические топологии

Основным представителем этого класса является перекрестное соединение (рис. 1.20).

Оно обеспечивает полную связность, т. е. каждый процессор может связаться с любым другим процессором или модулем памяти (как на рисунке). Пропускная способность сети при этом не уменьшается. Схемы с перекрестными переключениями могут использоваться и для организации межпроцессорных соединений. В этом случае каждый процессор имеет собственный модуль памяти (системы с распределенной памятью).

Для соединения *n* процессоров с *n* модулями памяти требуется n^2 переключателей. Такая зависимость ограничивает масштабируемость системы (обычно не более 256 узлов).

Многокаскадные сети

Многокаскадные (multistage) сети основаны на использовании 2x2 перекрестных переключателей — коммутаторов (рис. 1.21).

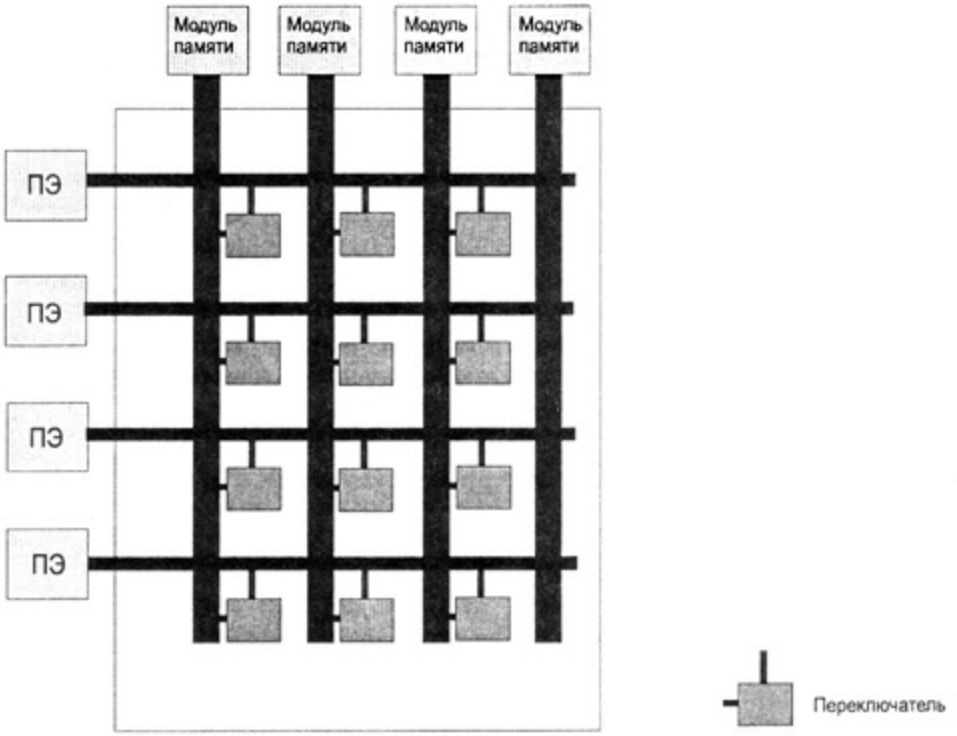


Рис. 1.20. *Схема перекрестного соединения*

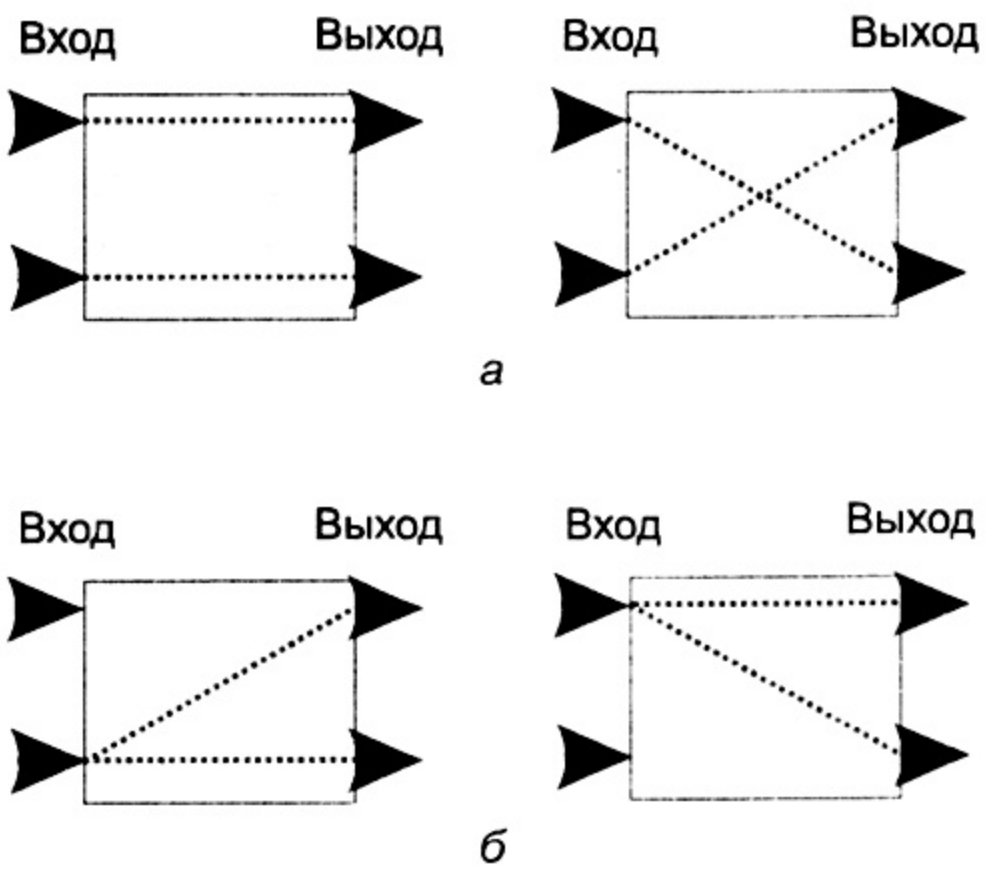


Рис. 1.21. *Переключатели без широковещательной рассылки (а) и с широковещательной рассылкой (б)*

В этом случае на одном конце соединения находятся процессоры, а на другом — процессоры или другие узлы. Между ними располагаются переключатели. При передаче данных от узла к узлу переключатели устанавливаются таким образом, чтобы обеспечить требуемое соединение. Очевидно, для этого требуется некоторое время — *время установки.* В зависимости от способа соединения имеются разные топологии динамических многокаскадных сетей. Вот некоторые из них:

- баньян (схема сети похожа на корни экзотического растения баньян — индийской смоковницы);
- куб;
- "дельта";
- триггер;
- "омега".

В качестве примера многокаскадной сети на рис. 1.22 приведена схема сети Бенеша 8x8. Она состоит из двух последовательно соединенных сетей баньян и является сетью с блокировкой, в которой один маршрут передачи данных может блокировать другие маршруты. Существуют и не блокирующие сети.

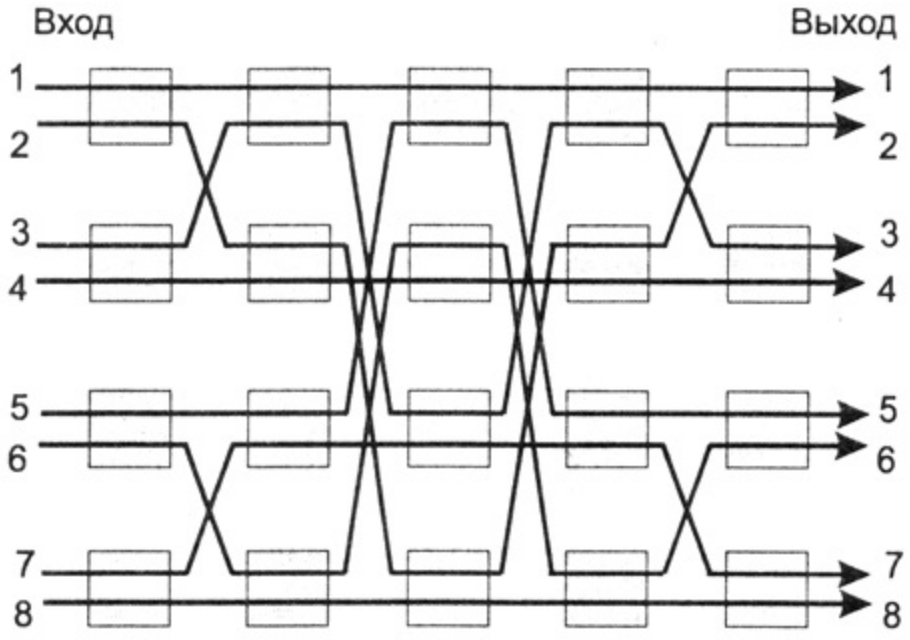


Рис. 1.22. *Сеть Бенеша*

На рис. 1.23 приведена схема соединения "омега".

Более устойчивую и эффективную работу обеспечивает сеть с коммутационной матрицей (рис. 1.24). Масштабируемость такого соединения не очень хорошая, т. к. добавление нового узла требует введения в схему дополнительно *2п* - 1 переключателей.

В случае сети с коммутационной матрицей каждый процессорный элемент может быть связан с любым другим. В отличие от сети с полносвязной топологией требуется меньше связей, но больше переключателей. Параллельно могут осуществляться несколько операций пересылки данных.

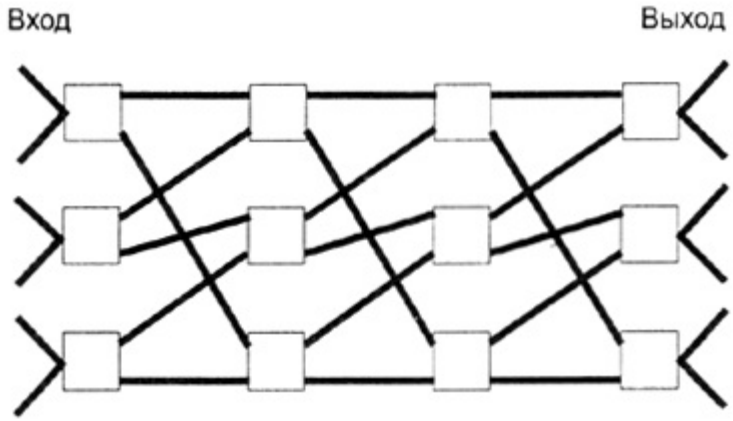


Рис. 1.23. *Соединение "омега"*

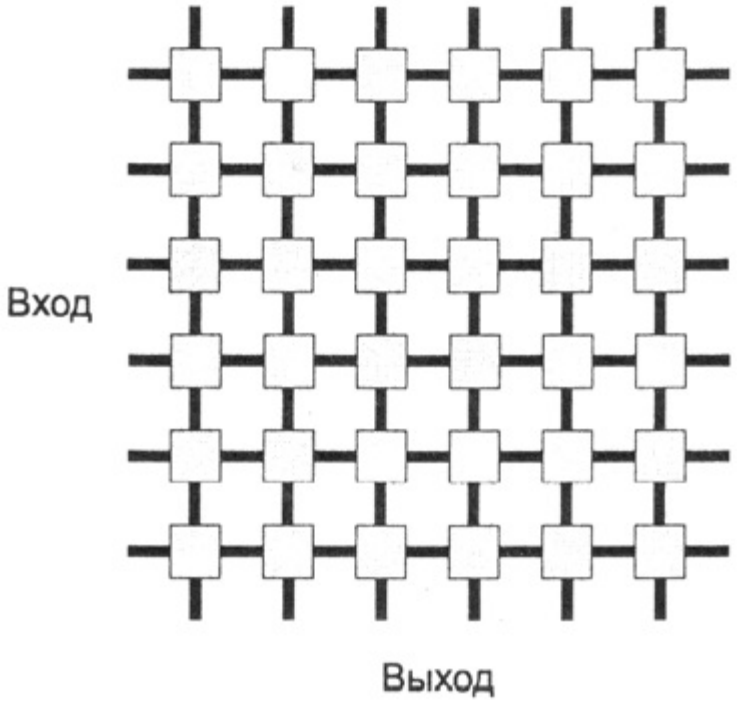


Рис. 1.24. *Соединение с коммутационной матрицей*

Важнейшими атрибутами системы коммуникаций являются стратегии управления, коммутации и синхронизации. Что касается управления, то здесь можно выделить две альтернативы: *централизованное управление* единым контроллером (модулем управления) и *распределенное управление.* Распределенное управление применяется в многокаскадных соединениях, где каждый узел принимает решение, как действовать с поступившим сообщением — оставить его себе или передать соседу. Другой вариант используется, например, в соединениях типа "звезда", где каждое сообщение пересылается в контроллер, который и определяет его дальнейшую судьбу.

Синхронизация тоже может быть глобальной, когда синхронизующая последовательность импульсов передается всем узлам вычислительной системы, но может быть и локальной, когда каждый узел имеет свой собственный генератор. Последний вариант называется *асинхронной работой.* Преимущество глобальной синхронизации, характерной для SIMD-компьютеров, заключается в более простой аппаратной и программной реализации, а асинхронные системы, чаще всего это MIMD-компьютеры, более гибкие.

Методы коммутации (переключения)

Важнейшим в организации работы коммуникационной сети является метод переключения, определяющий, как сообщения передаются от узла-источника к узлу-приемнику.

Сообщением называют логически завершенную порцию данных — пересылаемый файл, запрос на пересылку файла и т. д. Сообщения могут иметь любую длину. *Пакет*

представляет собой часть сообщения, его длина ограничена, хотя и может варьироваться. Пакеты перемещаются по сети независимо друг от друга. Каждый пакет содержит заголовок с информацией об адресе получателя и его номер, что необходимо для правильной "сборки" пакета и восстановления целого сообщения.

Выделяют три основных метода коммутации в коммуникационных сетях параллельных вычислительных систем:

- коммутация с промежуточным хранением ("хранение-и-передача" — Store-and-Forward);
- коммутация цепей;
- метод виртуальных каналов.

Исторически, одним из первых методов коммутации был метод с промежуточным хранением (метод коммутации пакетов). В этом случае сообщение полностью принимается на каждом промежуточном узле и только после этого отправляется дальше. Пересылка выполняется, как только освобождается очередной канал передачи, тогда сообщение отправляется на следующий узел. Целое сообщение разбивается на пакеты, которые при достижении очередного узла сохраняются в специальном буфере. Буферизация требует дополнительной памяти и затрат времени. Задержка передачи пропорциональна расстоянию между источником и адресатом. Данный метод применяется в ситуациях, когда время отклика не имеет значения.

При коммутации цепей (коммутации каналов) между источником и получателем сообщения устанавливается непрерывная цепь, состоящая из отдельных каналов связи, проходящих через промежуточные узлы. После этого выполняется передача данных. Коммутируемый канал устанавливается только на время соединения отправителя и адресата.

Уменьшить задержки при передаче данных позволяет метод виртуальных каналов. В этом случае пакеты накапливаются в промежуточных узлах только тогда, когда недоступен очередной канал связи. В противном случае, пересылка выполняется немедленно и без буферизации.

Схемы классификации архитектур параллельных компьютеров

С одной из схем классификации мы уже познакомились — это таксономия Флинна. Она является одной из наиболее распространенных схем классификации, но не единственной. Действительно, разве можно уложить все многообразие компьютерных архитектур в одну схему? Часто используется классификация, в которой за основу берется способ взаимодействия процессоров с оперативной памятью. В схеме Хандлера, или *эрлангерской схеме,* учитывается количество управляющих устройств и функциональных узлов, особенности устройства компьютера на уровне машинного слова. Более подробное описание этих и других таксономии можно найти в

специальной литературе, мы же кратко опишем две упомянутые схемы.

Классификация по способу взаимодействия процессоров с оперативной памятью

В этой схеме выделяют три основные группы архитектур:

- с разделяемой памятью;
- с распределенной памятью;
- *с* распределенно-разделяемой памятью.

Основным свойством систем с разделяемой памятью является то, что все процессоры системы имеют доступ к одной оперативной памяти, используя единое адресное пространство. Обычно главная память состоит из нескольких модулей памяти (их число не обязательно совпадает с числом процессоров).

В такой системе связь между процессорами выполняется с помощью разделяемых переменных. Этот тип параллельных компьютеров называют также компьютерами *с однородным доступом к памяти* и обозначают английской аббревиатурой UMA (Uniform Memory Access), поскольку параметры доступа к модулям памяти для всех процессоров одинаковы.

Преимуществом компьютеров с разделяемой памятью является удобство программирования для них, поскольку все данные доступны всем процессорам, и не надо заботиться о пересылках данных. Программист не должен думать и о синхронизации, потому что синхронизацию обеспечивает сама система. Однако на компьютерах с разделяемой памятью сложно достичь параллелизма высокого уровня, поскольку большинство таких систем содержат менее 64 процессоров. Это ограничение следует из плохой масштабируемости централизованной памяти и системы коммуникаций.

В случае компьютера с распределенной памятью каждый процессор имеет собственную оперативную память. Глобального адресного пространства в этом случае уже нет. Коммуникации и синхронизация процессоров осуществляются с помощью обмена сообщениями по коммуникационной сети.

В отличие от систем с разделяемой памятью системы с распределенной памятью очень хорошо масштабируются, поскольку в этом случае исключены конфликты по доступу к памяти. В результате могут создаваться системы с высокой степенью параллелизма (MPP — Massively Parallel Processors), состоящие из сотен и тысяч процессоров. Типичными представителями систем с распределенной памятью являются кластеры рабочих станций, объединенные коммуникационной сетью достаточно дешевой, но обеспечивающей приемлемую скорость обмена данными (Ethernet, Myrinet и др.).

В системах с распределенно-разделяемой памятью используются преимущества обоих подходов. Это относительная простота программирования с одной стороны, хорошая масштабируемость с другой. Каждый процессор имеет собственную локальную память, но, в отличие от архитектуры с распределенной памятью, все модули памяти образуют единое адресное пространство, т. е. каждая ячейка памяти имеет адрес, единый для всей системы.

Схема Хандлера (эрлангерская схема)

В эрлангерской схеме, предложенной Хандлером в 1977 году, параллельные компьютеры классифицируются по степени параллелизма и конвейеризации на трех уровнях абстракции:

- уровень выполнения программы/процесса;
- уровень выполнения команд;
- уровень подкоманд (уровень поразрядной обработки).

Описание компьютера дается тремя парами значений для соответствующих уровней параллелизма (табл. 1.2):

(Kx K', DxD', W* W').

Таблица 1.2. *Схема Хандлера классификации компьютерных архитектур*

Уровень параллелизма	**Параллелизм**	**Конвейеризация**
Программа/ процесс	*K*— количество устройств управления	*K'*— количество сегментов в макроконвейере
Машинные команды	D— количество АЛУ в каждом устройстве управления	D'— количество сегментов в конвейере
Подкоманды	*W*— количество элементарных логических схем в каждом АЛУ	*W*— количество сегментов в конвейере

Например, по этой классификации для компьютера Intel Paragon XP/S с 1840 узлами получаем:

(1840 x 2, 1 x 2, 64 x 3),

потому что каждый из *K*— 1840 узлов состоит из двух процессоров Intel i860 *(K'*— 2). Каждый процессор может одновременно выполнять две команды *(D'*= 2), разрядность функциональных узлов 64 бита *(W*= 64), глубина конвейера равна трем *(W*= 3).

Основные типы архитектур высокопроизводительных вычислительных систем

SIMD-архитектуры с разделяемой памятью

SIMD-архитектуры с разделяемой памятью объединяют векторные процессоры, VLIW-процессоры и др. В векторных процессорах часто нет кэшпамяти, поскольку при векторной обработке использование преимуществ кэш-памяти затруднено и даже может отрицательно сказываться на производительности (например, вследствие частых переполнений кэш-памяти).

Современные векторные процессоры используют векторные регистры, а не прямое обращение к памяти. Разница в быстродействии небольшая, но векторные регистры обеспечивают большую гибкость программирования. Если отношение числа арифметических операций к числу операций обращения к памяти мало, потери производительности будут большими, вследствие относительно низкой скорости обмена с памятью. Бороться с потерями такого рода можно, наращивая пропускную способность каналов доступа к памяти, но это сравнительно дорогой способ, поэтому обычно используются другие, компромиссные способы.

Устройство векторных узлов может варьироваться в широких пределах. Каждый узел обычно содержит несколько векторных функциональных устройств. Среди них конвейеры, реализующие функции доступа к памяти, несколько узлов целочисленной арифметики и логических операций, сложения и умножения с плавающей точкой (раздельные и комбинированные), конвейер операций маскирования.

SIMD-машины с распределенной памятью

SIMD-машины с распределенной памятью называют также *матричными процессорами.* В системах с такой архитектурой все процессоры в один и тот же момент времени выполняют одну команду. Имеется управляющий процессор, который передает команды процессорам массива. Обычно есть и *"фронтальный"* (front-end) процессор, которому отводится функция связи с пользователями, а также выполнение тех операций, которые не могут быть выполнены массивом процессоров (например, операции ввода/вывода). Коммуникационная сеть в таких системах имеет топологию двумерной решетки или, по крайней мере, включает ее в качестве составной части

подсистем коммуникации. С помощью логических условий некоторые процессоры можно исключать из процесса обработки данных, переводя их в режим ожидания.

Одной из проблем, связанных с работой систем с распределенной памятью, является обработка ситуаций, когда данные, необходимые для продолжения работы определенного процессора, находятся в локальной памяти другого процессора. Выборка необходимых операндов и их пересылка по сети требует значительного времени, поэтому SIMD-машины с распределенной памятью часто специализированы для решения вполне определенных задач -в этом случае можно добиться наиболее полного использования присущего задаче параллелизма. Имеются специализированные системы для расчетов методом Монте-Карло (таких, где не требуется большой обмен данными между процессорами), обработки изображений и сигналов и т. д.

Управляющий процессор в рассматриваемых системах должен не только формировать поток команд и направлять его процессорам массива, но и решать, какие операции следует "поручить" фронтальному процессору и связанными с ним периферийными устройствами. Такими операциями могут быть, например, печать результатов или их сохранение на магнитном носителе информации. Впрочем, операции ввода/вывода в подобных системах часто реализованы независимо от фронтальной машины, с помощью специализированных устройств.

MIMD-машины с разделяемой памятью

Основной проблемой в вычислительных системах такого рода является связь процессоров между собой и с памятью. С увеличением числа процессоров суммарная пропускная способность каналов доступа к памяти в идеале должна расти линейно с ростом числа процессоров, а коммуникации между процессорами должны быть, по возможности, прямыми, минуя медленный промежуточный этап обращения к оперативной памяти. Обеспечить полную связь сложно, поскольку число коммуникаций растет пропорционально второй степени числа процессоров.

Тип соединения важен и для синхронизации выполнения задач на разных процессорах. Большинство векторных процессоров имеют специальные коммуникационные регистры, с помощью которых решается задача синхронизации с другими процессорами. Реже используется синхронизация через разделяемую память. Этот способ более медленный и он подходит для ситуаций, в которых потребность в синхронизации возникает редко. В системах с использованием шины для синхронизации работы процессоров применяют специальную шину, отдельную от шины данных.

MIMD-машины с распределенной памятью

Это наиболее быстро растущий класс высокопроизводительных компьютеров.

Программирование для таких систем сложнее, чем для MIMD-машин с разделяемой памятью. В матричных процессорах основной структурой данных являются массивы, которые хорошо отображаются на структуру машины. В этом случае распределение данных является "прозрачным" для программиста и в значительной степени автоматизировано системным программным обеспечением. В системах с распределенной памятью, напротив, программист сам должен позаботиться о распределении данных между процессорами и обмене данными между ними. Решение проблемы декомпозиции данных и задачи, разработка эффективного параллельного алгоритма часто оказываются непростым делом.

Значительным преимуществом систем данного класса является отсутствие проблемы ограниченной скорости обмена данными и хорошая масштабируемость. Не так остро стоит и проблема согласования скорости работы процессора и памяти.

Есть и недостатки. Например, межпроцессорные соединения гораздо более медленные, чем в системах с разделяемой памятью. Синхронизация задач в таком случае почти на порядок замедляет работу системы.

В MIMD-машинах с распределенной памятью часто используется топология "гиперкуб". Теоретически, в сетях с топологией гиперкуба можно имитировать другие топологии, такие как "дерево", "кольцо", "решетка" и др. Другой эффективной топологией соединения является "толстое дерево". Простое дерево имеет тот. существенный недостаток, что вблизи его "корня", где пропускная способность относительно невелика, увеличивается концентрация циркулирующих сообщений. В топологии "толстое дерево" используется дублирование связей с высокими уровнями дерева. Пример такой структуры приведен на рис. 1.25.

У "толстого дерева" пропускная способность у "корня" выше, чем у "листьев".

Достаточно часто используются коммуникационные матрицы, однокаскад-ные для систем с небольшим числом процессоров (порядка 64) и многокаскадные для систем с большим числом процессоров.

Процессоры используются любые, обычно это RISC-процессоры и векторные процессоры. Часто применяется также специальный коммуникационный процессор.

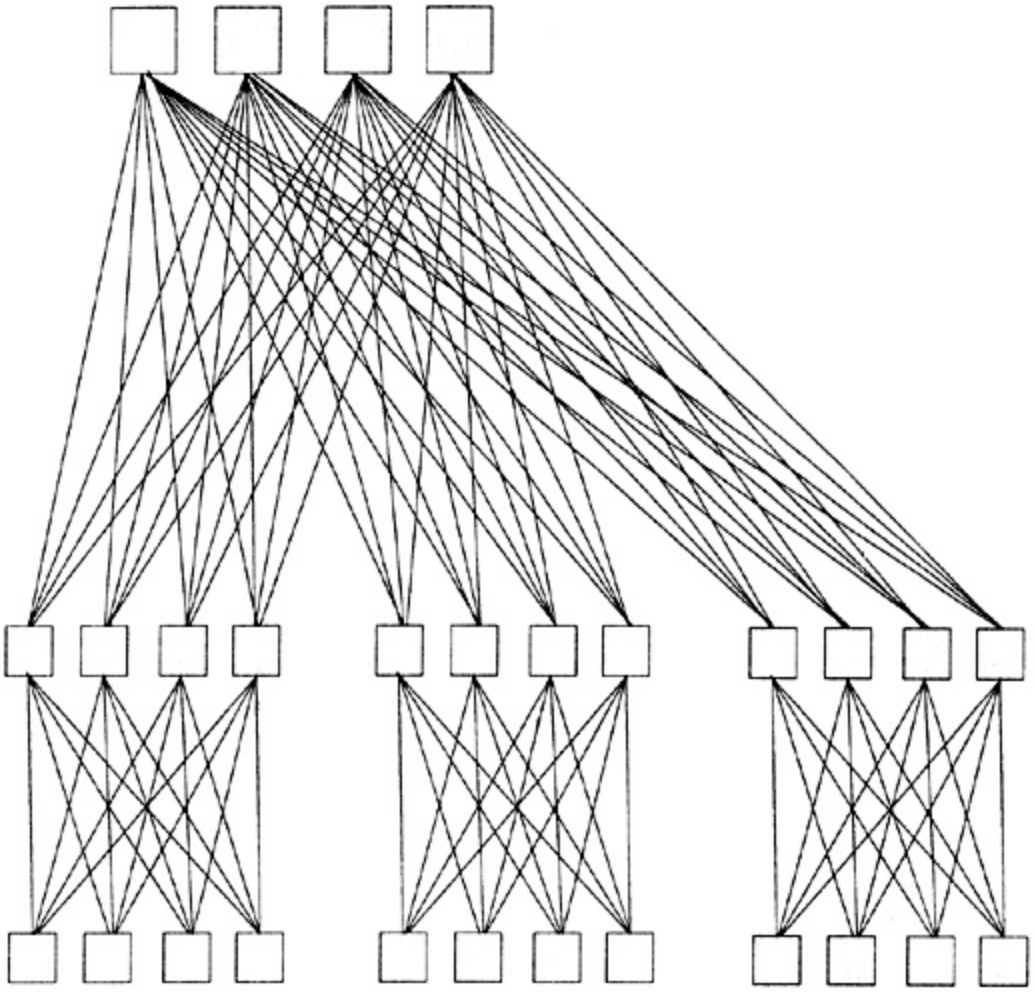


Рис. 1.25. *Структура соединения "толстое дерево"*

Архитектура ccNUMA

В настоящее время наблюдается тенденция создания комбинированных систем (ccNUMA — Cache Coherent Non-Uniform Memory Access), в которых базовой архитектурной единицей является многопроцессорная конфигурация, составленная из небольшого числа RISC-процессоров (до 16) с одно-каскадным соединением. Это система с "сильной связью". Из таких "кирпичиков" строится симметричная многопроцессорная кластерная система со 'слабой связью" посредством, например, многокаскадной сети. Системы такого рода можно отнести к MIMD-системам с разделяемой памятью, поскольку при необходимости все процессоры имеют доступ к единому адресному пространству. Необходимый элемент данных может располагаться в любом модуле памяти, однако логически он находится в едином адресном пространстве. Неоднородность заключается в том, что время выборки различных данных может быть разным, поскольку физически они могут располагаться в разных модулях памяти. В системах такого типа приходится решать проблему кэш-когерентности.

Кластеры рабочих станций

Кластеры рабочих станций представляют собой совокупность рабочих станций, соединенных в локальную сеть, обычно, в масштабе отдела, факультета или института. Такой кластер можно считать вычислительной системой с распределенной памятью и распределенным управлением. Кластерная система, при относительно невысокой стоимости, может обладать производительностью, сравнимой с производительностью суперкомпьютеров. Ведь часто оказывается, что рабочие станции, роль которых могут играть персональные компьютеры, закуплены и установлены и не всегда загружены работой, так почему бы не превратить их в виртуальный вычислительный комплекс? Необходимое для работы параллельного кластера программное обеспечение — бесплатное, в том числе и операционная система. Еще одним из преимуществ такой, в общем случае гетерогенной (разнородной) вычислительной системы может быть то, что отдельные части параллельных программ могут выполняться на наиболее подходящих для этого компьютерах.

Для того чтобы построить работоспособный параллельный кластер, необходимо решить ряд проблем. Прежде всего, следует иметь в виду, что довольно часто в сеть объединяются компьютеры различных фирм-производителей, имеющие разную архитектуру, работающие под управлением разных операционных систем, имеющих разные файловые системы и т. д. То есть возникает проблема совместимости. Имеется и проблема доступа, т. к. для входа в любой компьютер может потребоваться разрешение работать на нем. Может быть и так, что время счета измеряется днями, а то и неделями, а некоторые из рабочих станций кластера время от времени должны быть загружены своей собственной работой. Следовательно, особенно важным оказывается управление такой вычислительной системой, динамичной и специфической.

Некоторые из перечисленных проблем могут быть решены административным образом, для решения других требуется специальное программное обеспечение, реже специальная аппаратура. В качестве примера приведем систему управления гетерогенными кластерами рабочих станций Condor. Эта система позволяет пользователю сохранить за собой полный контроль за своей собственной рабочей станцией и не требует от него наличия прав доступа к другим рабочим станциям. Она также обеспечивает достаточно гибкое управление процессом выполнения программ. Важно еще и то, что эта система бесплатная. Есть и коммерческие системы — Codine, LoadLeveler и некоторые другие.

Кластеры рабочих станций обычно называют *Беовульф-кластерами* (Beowulf cluster — по одноименному проекту). Типичный современный Беовульф-кластер представляет собой кластер рабочих станций, связанных локальной сетью Ethernet и обычно работающих под управлением ОС Linux, хотя в настоящее время разнообразие Беовульф-кластеров достаточно велико.

Первый Беовульф-кластер появился в 1994 году. При организации IEEE даже был

создан специальный комитет по кластерным системам - TFCC (Task Force *-on* Cluster Computing). Интерес к коммерческому выпуску кластерных систем возник и у крупных производителей компьютеров.

Взаимодействие компьютеров в кластере может быть организовано с помощью сети Ethernet 10 или 100 Мбит/с. Это наиболее экономичное решение, но не самое эффективное (время отклика около 100 микросекунд). Сеть на основе Gigabit Ethernet имеет пропускную способность на порядок больше. Имеются и другие возможности, это сети Myrinet, Giganet cLAN и SCI (время отклика 10—20 микросекунд). Наибольшая пропускная способность у сети SCI (до 500 Мбит/с), поэтому данный вариант часто используется в кластерах. Кластерные системы можно отнести к MIMD-системам с распределенной памятью.

Мультипроцессорные и мультикомпьютерные системы

MIMD-компьютеры с разделяемой памятью иногда называют *мультипроцессорными системами.* В некоторых мультипроцессорных системах отсутствует общая разделяемая память, вместо нее у каждого процессорного элемента имеется собственная локальная память, но, тем не менее, каждый процессорный элемент имеет доступ к локальной памяти любого процессорного элемента. В этом случае говорят о наличии глобального адресного пространства. Такая организация памяти называется *распределенно-разделяемой памятью.* Передача данных между процессорными элементами осуществляется через разделяемую память — один ПЭ может произвести запись в ячейку памяти, а все остальные ПЭ могут это значение использовать. С точки зрения программиста, коммуникации реализуются посредством разделяемых переменных, т. е. переменных, доступ к которым имеют все параллельные процессы. Примерами вычислительных мультипроцессорных систем могут служить: Cray X-MP, Cray Y-MP, Cray C90, Cray-3.

MIMD-компьютеры с распределенным адресным пространством, так что каждый ПЭ имеет собственную, локальную оперативную память, "невидимую" другими ПЭ, иногда называются *мулътикомпьютерами.* Взаимодействие между ПЭ реализуется обменом сообщениями, которые передаются по коммуникационной сети. С точки зрения программиста это означает, что для взаимодействия между ПЭ используются не разделяемые переменные, а операции пересылки и приема, а также каналы. Поскольку в данном случае нет конкуренции между процессорами за доступ к оперативной памяти, количество ПЭ не ограничено объемом оперативной памяти. Результирующее быстродействие вычислительной системы определяется скоростью работы сети. Примерами мультикомпьютеров являются — nCube (8192 ПЭ, гиперкуб), Cray T3E, кластеры рабочих станций и др.

Симметричные многопроцессорные системы

Симметричные многопроцессорные системы содержат несколько процессоров в

одном компьютере. Каждый процессор имеет доступ к общему набору модулей памяти и работает как процессор общего назначения, который может выполнять любой процесс. Распределение процессов между процессорами осуществляет операционная система, причем алгоритм распределения должен обеспечивать равномерную загрузку процессоров. При этом процессы могут перемещаться на освободившиеся процессоры, что является непростой задачей, решение которой возлагается на операционную систему. Примерами симметричных многопроцессорных систем являются UltraSPARC, IBM RS/6000.

Примеры архитектур суперкомпьютеров

Рынок высокопроизводительных вычислительных систем очень динамичен — регулярно появляются новые, все более мощные системы, с производства снимаются старые. Значительная часть современных высокопроизводительных систем — это кластеры симметричных многопроцессорных узлов, построенных из RISC-процессоров, соединенных быстрой сетью. Бурно развивается и рынок Беовульф-кластеров, различные фирмы предлагают сконфигурированные и отлаженные кластеры рабочих станций. Такие решения пользуются популярностью благодаря относительной дешевизне, не идущей в ущерб надежности.

В настоящее время ведущими производителями высокопроизводительных вычислительных систем являются: Compaq (продолжение линии AlphaServer SC), Cray Inc. (Cray SV-1, SV-2, MTA-2), Hewlett Packard (HP SuperDome), IBM (ASCI Blue Pacific), SGI (Origin), SRC (SRC-6 и специализированные вычислительные системы). В скобках указаны выпускаемые в настоящее время высокопроизводительные вычислительные системы.

Краткий перечень современных высокопроизводительных вычислительных систем включает: Cambridge Parallel Processing Gamma II, C-DAC Param 10000 OpenFrame, серию Compaq GS, серию Compaq AlphaServer SC, Cray SV1, Cray T3E, Cray MTA-2, Fujitsu AP3000, Fujitsu/Siemens PRIMEPOWER, серию Fujitsu VPP5000, Hitachi SR8000, HP 9000 SuperDome, IBM RS/6000 SP, NEC Cenju-4, NEC SX-5, Quadrics Apemille, серию SGI Origin 3000, Sun E10000 Starfire.

Во многих вычислительных центрах продолжают использоваться и устаревшие или снятые с производства вычислительные системы. Их неполный список приведен в табл. 1.3.

Таблица 1.3. *Устаревшие и снятые с производства высокопроизводительные системы*

	Год	**Год завер-**		**Пиковая произво-**	**Причина**

Система	начала выпуска	шения выпуска	Тип	дительность Гфлоп/с	снятия с производства
Alex AVX 2	1992	1997	RISC-мультипроцессор с распределенной памятью	3,84	Система устарела
Alliant FX/2800	1989	1992	Векторно-параллельный компьютер с разделяемой памятью	1,12	Уход с рынка фирмы-производителя
Avalon A12	1996	2000	RISC-мультипроцессорная система с распределенной памятью (до 1680 процессоров)	1300	Уход с рынка фирмы-производителя
BBN TC2000		1990	Параллельная система с виртуальной разделяемой памятью (до 512 процессоров)	1000	Уход с рынка фирмы-производителя
Convex SPP-1000 /1200/1600	1995	1996	Многопроцессорная RISC-система с распределенной памятью (до 1 28 процессоров)	25,6	Замена новой системой HP 9000 SuperDome
Cray-2	1982	1992	Векторно-параллельная система (до 4 процессоров) с разделяемой памятью	1,95	Уход с рынка фирмы-производителя
Cray-3	1993	1995	Векторно-параллельная система (до 16 процессоров) с разделяемой памятью	16	Уход с рынка фирмы-производителя

Cray T3D	1994	1996	Многопроцессорная RISC-система с распределенной памятью (до 2048 процессоров)	307	Замена моделью Cray T3E
Cray J90	1994	1998	Векторно-параллельная система с виртуальной разделяемой памятью (до 32 процессоров)	6,4	Замена моделью Cray SV1
Cray Y-MP C90	1994	1996	Векторно-параллельная система с разделяемой памятью (до 16 процессоров)	16	Замена моделью Cray T90
Cray T90	1995	1998	Векторно-параллельная система с разделяемой памятью (до 32 процессоров)	58	Замена моделью Cray SV1

Fujitsu AP 1000	1991	1996	Многопроцессорная RISC-система с распределенной памятью (до 1024 процессоров)	5	Замена моделью Fujitsu AP3000
Hitachi S-3600 series	1994	1996	Однопроцессорная векторная система	2	Замена моделью Hitachi SR8000
Hitachi SR2201	1996	1998	Многопроцессорная RISC-система с распределенной памятью (до 1024 процессоров)	307	Замена моделью Hitachi SR8000
			Векторно-параллельная		

HP/Convex C4600	1994	1997	система с разделяемой памятью (до 4 процессоров)	3,2	Снятие серии C с производства
HP Exemplar V2600	1999	2000	Многопроцессорная RISC-система с распределенной памятью (до 1 28 процессоров)	291	Замена моделью HP 9000 SuperDome
IBM SP1	1992	1994	Многопроцессорная RISC-система с распределенной памятью (до 64 процессоров)	8	Замена моделью IBM RS/6000 SP
Intel Paragon XP	1992	1996	Многопроцессорная RISC-система с распределенной памятью (до 4000 процессоров)	300	Уход с рынка фирмы-производителя (в настоящее время IBM выпускает только исследовательские системы)
Kendall Square Research KSR2	1992	1994	Многопроцессорная RISC-система с разделяемой памятью (до 1088 процессоров)	400	Уход с рынка фирмы-производителя
Parsytec CC	1996	1998	Многопроцессорная RISC-система с распределенной памятью	-	Уход с рынка фирмы-производителя
Thinking Machine Corporation CM-5	1991	1996	SIMD-параллельный компьютер с топологией гиперкуб (до 64 000 процессоров)	31	Изменение профиля деятельности фирмы-производителя
			Многопроцессорная система с		Замена моделью

SGI Origin 2000	1996	2000	разделяемой памятью (до 1 28 процессоров)	102,4	SGI Origin 3000

Рассмотрим архитектуру и технические характеристики нескольких высокопроизводительных систем.

Архитектура суперкомпьютера NEC SX-4

NEC SX-4 1996 года выпуска (NEC — японская фирма, один из крупнейших производителей суперкомпьютеров), является параллельно-векторным компьютером с двухступенчатой архитектурой. Основным строительным блоком является узел, содержащий до 32 процессоров с общей пиковой производительностью 64 Гфлоп/с. Процессоры в узле соединяются посредством коммутационной матрицы с 1024 банками памяти общим объемом до 16 Гбайт. Кроме того, каждый узел (рис. 1.26) поддерживает до 4 модулей расширенной памяти (XMU — Extended Memory Unit) с суммарным объемом до 32 Гбайт оперативной памяти и 4 процессорами ВВ (IOP — I/O Processor).

Второй уровень — архитектура с распределенной памятью. До 16 узлов, а это 512 процессоров с общей пиковой производительностью 1 Тфлоп/с, объединяются с помощью коммутационной матрицы. С аппаратной точки зрения это кластер, для программиста же система является однородной.

Каждый процессор состоит из векторного модуля, скалярного модуля и модуля обработки команд. Тактовая частота относительно невелика — 125 МГц. Векторный модуль развивает максимальное быстродействие 2 Гфлоп/с. Скалярный модуль имеет обычную суперскалярную архитектуру с той же тактовой частотой, что и векторный модуль, и максимальное быстродействие 250 Мфлоп/с. Процессор поддерживает форматы представления чисел с плавающей точкой IEEE, Cray и IBM, а пользователь может выбрать тип используемой арифметики во время трансляции программы. Это удобно, поскольку пользователи из академической среды предпочитают формат IEEE, а пользователи "промышленные" используют форматы Cray или IBM.

Все вышеперечисленные особенности архитектуры обеспечивают хорошую масштабируемость системы и медленное падение скорости обработки пользовательских задач даже при существенной загрузке системы. Система работает под управлением операционной системы Super-UX (64-разрядная ОС из семейства UNIX с поддержкой многопроцессорности и многопоточ-ности).

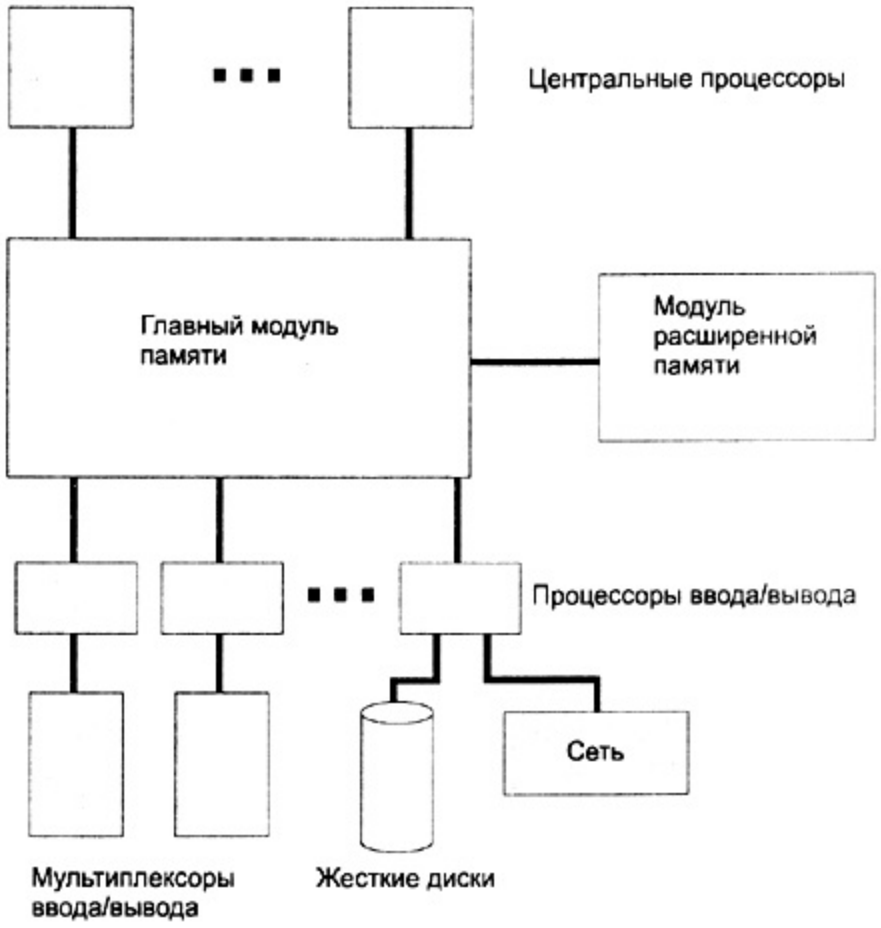


Рис. 1.26. *Схема узла NEC SX-4*

Compaq AlphaServer SC

Эта система 1999 года выпуска представляет собой симметричную многопроцессорную систему с распределенной памятью, RISC-архитектурой и коммутационной матрицей. Программное обеспечение — ОС Digital UNIX, трансляторы для языков FORTRAN, High-Performance FORTRAN (HPF), C и C++. Тактовая частота 833 МГц, максимальный размер оперативной памяти 2 Тб и максимальное число процессоров 512. Скорость обмена между процессором и памятью 1,33 Гбайт/с и между ПЭ 210-Мбайт/с. Теоретическое максимальное быстродействие каждого процессора 1,67 Гфлоп/с, а всей системы 853 Гфлоп/с.

Каждый узел представляет собой 4-процессорную симметричную систему (Compaq ES40). Процессоры — Alpha 21264a (EV67). В каждом узле используется коммуникационная матрица с пропускной способностью 5,2 Гбайт/с и разделяемая память. Узлы собраны в кластер и соединяются сетью QsNet (SQW Ltd.) с топологией "толстого дерева".

Если для выполнения параллельной программы достаточно не более четырех процессоров, можно использовать модель программирования с разделяемой памятью (например, с использованием OpenMP). Если же необходимое количество процессоров превышает 4, применяется модель с передачей сообщений (с

использованием MPI, PVM, HPF). Быстродействие системы при решении систем линейных алгебраических уравнений (порядок системы 200 000) около 500 Гфлоп/с.

Архитектура суперкомпьютера Cray SX-6

В 2001 году фирма Cray Inc. совместно с NEC выпустила на рынок масштабируемый, параллельно-векторный суперкомпьютер Cray SX-6, один из наиболее мощных современных суперкомпьютеров. Cray SX-6 представляет собой симметричную мультипроцессорную систему, строительными блоками которой являются параллельно-векторные процессорные узлы. Узлы состоят из нескольких (от двух до восьми) векторных процессоров, максимальное быстродействие каждого из них составляет 8 Гфлоп. Каждый процессор имеет доступ к разделяемой высокопроизводительной памяти объемом от 16 до 64 Гбайт и скоростью передачи данных 256 Гбайт/с. Конфигурации, состоящие из нескольких узлов, развивают максимальное быстродействие до 8 Тфлоп (один триллион операций с плавающей точкой в секунду), имеют оперативную память объемом 8 Тбайт и скорость обмена данными с устройствами ВВ 800 Гбайт/с.

Здесь, таким образом, используется комбинированная схема организации памяти — разделяемая внутри одного узла и распределенная в многоузельной конфигурации. Это позволяет, при необходимости, использовать особенности обоих типов организации оперативной памяти. Скорость обмена данными с оперативной памятью велика, а время задержки мало. В много-узельной конфигурации допускается динамическое изменение виртуальной многопроцессорной конфигурации путем включения в нее нового узла или удаления одного из уже имеющихся. Система работает под управлением ОС Super-UX.

Cray T3E (1350)

Это мультипроцессорная вычислительная система 2000 года выпуска, с распределенной памятью построена из RISC-процессоров. Топология коммуникационной сети — трехмерный тор. Операционная система UNICOS/mk (ОС UNIX с микроядром). Трансляторы для языков FORTRAN, HPF, C/C++. Тактовая частота 675 МГц. Количество процессоров от 40 до 2176.

Максимальный объем оперативной памяти для каждого узла 512 Мбайт и максимальное быстродействие 2938 Гфлоп/с. В отличие от предшественника — Cray T3D, данной системе не требуется фронтальный компьютер.

В системе используется процессор Alpha21164A, однако, при необходимости, его несложно заменить другим, например, более быстродействующим процессором. Каждый процессорный элемент содержит центральный процессор, модуль памяти и коммуникационный узел для связи с другими процессорами. Пропускная способность канала связи между процессорами 325 Мбайт/с.

Поддерживаются модели программирования MPI, PVM, HPF, собственная библиотека обмена сообщениями Cray shmem. Быстродействие, полученное при решении систем линейных алгебраических уравнений, достигает 1,12 Тфлоп/с.

Cray MTA-2

Это мультипроцессорная система с распределенной памятью, 2001 года выпуска. Работает под управлением ОС BSD. Количество процессоров до 256. В данной системе используется многопоточная архитектура. При выполнении программы поток команд разбивается на части, которые могут обрабатываться одновременно. Если, например, обращение к памяти в каком-либо из потоков не может быть выполнено, этот поток приостанавливается, а вместо него активизируется другой поток. Коммуникационная сеть представляет собой трехмерный куб. Каждый узел имеет собственный порт ввода/вывода.

Параллелизм в многопоточной архитектуре выявляется и реализуется автоматически, однако в системе Cray MTA-2 могут использоваться и явные модели параллельного программирования.

Вопросы и задания для самостоятельной работы

1. Что такое архитектура вычислительной системы?

2. Перечислите архитектуры вычислительных систем согласно классификации Флинна.

3. Опишите работу конвейера данных и конвейера команд.

4. Приведите классификацию высокопроизводительных ЭВМ по архитектуре подсистем оперативной памяти. Какие достоинства, какие недостатки есть у каждого из этих классов вычислительных систем?

5. Что называется топологией коммуникационной сети вычислительной системы? 6. Чем характеризуется коммуникационная сеть вычислительной системы?

7. Приведите пример статической топологии коммуникационной сети. Охарактеризуйте ее.

8. Приведите пример динамической топологии коммуникационной сети. Охарактеризуйте ее.

9. Дайте описание RISC-архитектуры процессора.

10. Дайте описание CISC-архитектуры процессора.

11. Дайте описание VLIW-архитектуры процессора.

12. Дайте характеристику Беовульф-кластеру как виртуальной многопроцессорной вычислительной системе.

13. Каким образом основные компоненты архитектуры вычислительной системы влияют на ее производительность?

14. В чем заключается проблема кэш-когерентности? Какие существуют способы ее решения?

15. Что такое суперскалярная архитектура?

16. Попробуйте "сконструировать" (разумеется, теоретически) вычислительную систему, которая могла бы обеспечить максимальное быстродействие при решении ваших прикладных задач. Это могут быть, например, задачи матричной алгебры, задачи сортировки, статистическая обработка больших массивов данных, аэродинамика, криптография и т. д.

ГЛАВА 2.

Особенности программирования параллельных вычислений

В этой главе мы познакомимся с основными принципами параллельного программирования. Любой программист начинает свою программистскую жизнь с создания последовательных программ. Последовательная модель программирования является традиционной. Но существуют и другие модели программирования. Одной из них является параллельная модель. Ее особенности, достоинства и связанные с ней проблемы обсуждаются в данной главе.

Разработка параллельной программы для программиста, привыкшего к последовательной модели программирования, может оказаться на первых порах непростым делом. Ведь в этом случае придется задуматься не только о привычных вещах, таких как структуры данных, последовательность их обработки, интерфейсы, но и об обменах данными между подзадачами, входящими в состав параллельного

приложения, об их согласованном выполнении и т. д. Однако, приобретя навык параллельного программирования, начинаешь понимать, что параллельное программирование — это… просто. Иногда написать параллельную программу для обработки массива данных оказывается проще, чем последовательную.

Последовательная и параллельная модели программирования

Совокупность приемов программирования, структур данных, отвечающих архитектуре гипотетического компьютера, предназначенного для выполнения определенного класса алгоритмов, называется *моделью программирования.* Напомним, что *алгоритм* — это конечный набор правил, расположенных в определенном логическом порядке и позволяющий исполнителю решать любую конкретную задачу из некоторого класса однотипных задач.

Описание алгоритма является иерархическим, его сложность зависит от степени его детализации. На самом верхнем уровне иерархии находится задача в целом. Ее можно считать некоторой обобщенной операцией. Нижний уровень иерархии включает примитивные операции, машинные команды.

Алгоритм можно представить в виде диаграммы — *информационного графа.* Информационный граф описывает последовательность выполнения операций и взаимную зависимость между различными операциями или блоками операций. Узлами информационного графа являются операции, а однонаправленными дугами — каналы обмена данными (рис. 2.1). Понятие операции может трактоваться расширенно. Это может быть оператор языка, но может быть и более крупный блок программы.

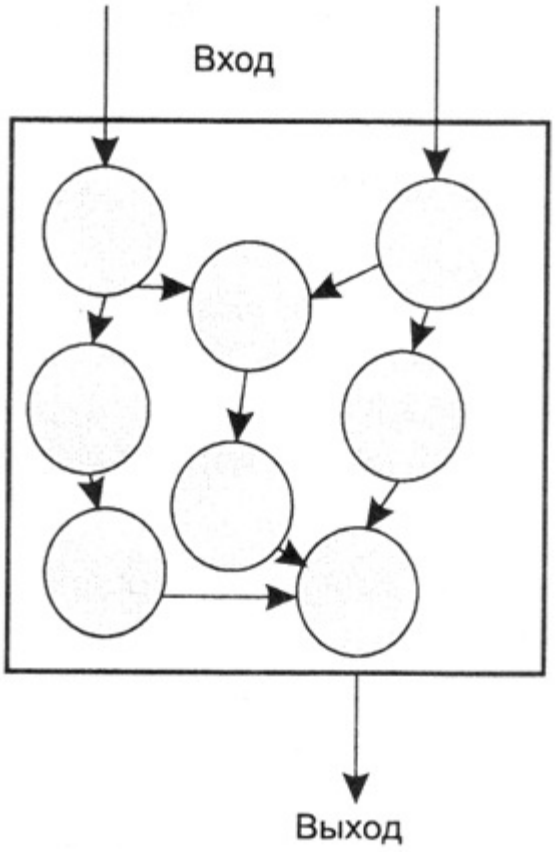


Рис. 2.1. *Информационный граф алгоритма*

В информационном графе каждый узел задается парой *(n, s),* где *n* — имя узла, *s* — его размер. Размер узла определяется количеством простейших операций, входящих в его состав. Дуги характеризуются значениями (v, *d),* где *v* — пересылаемая переменная, а *d* — время, затрачиваемое на ее пересылку. Слияние нескольких узлов в один (упаковка узлов) уменьшает степень детализации алгоритма, но увеличивает количество дуг, выходящих из вершины графа. Упаковка позволяет скрыть несущественные на данном этапе разработки коммуникации и снизить затраты на их планирование.

На рис. 2.2 и 2.3 представлены два предельных случая информационного графа. Первый из них соответствует последовательной, а второй — параллельной модели вычислений.

Традиционной считается *последовательная модель программирования.* В этом случае в любой момент времени выполняется только одна операция и только над одним элементом данных. Последовательная модель универсальна. Ее основными чертами являются применение стандартных языков программирования (для решения вычислительных задач это, обычно, FORTRAN 90/95 и C/C++), хорошая переносимость программ и невысокая производительность.

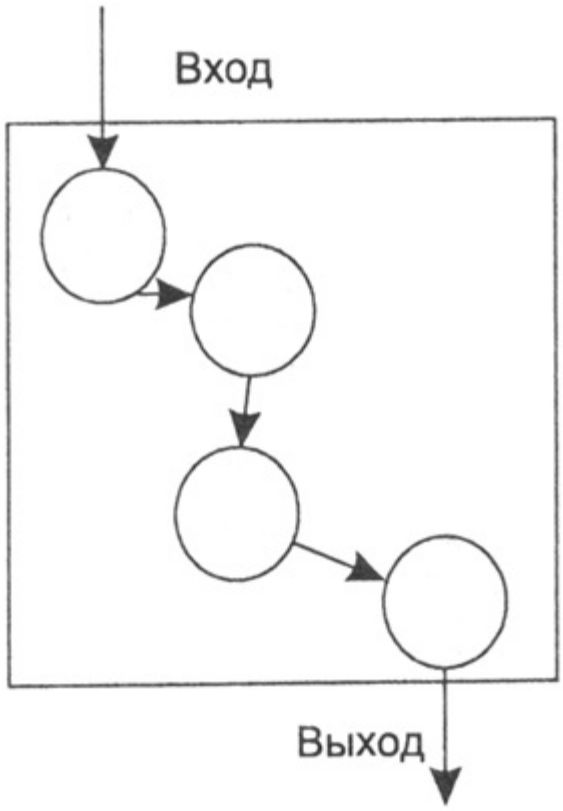


Рис. 2.2. *Полностью последовательный алгоритм*

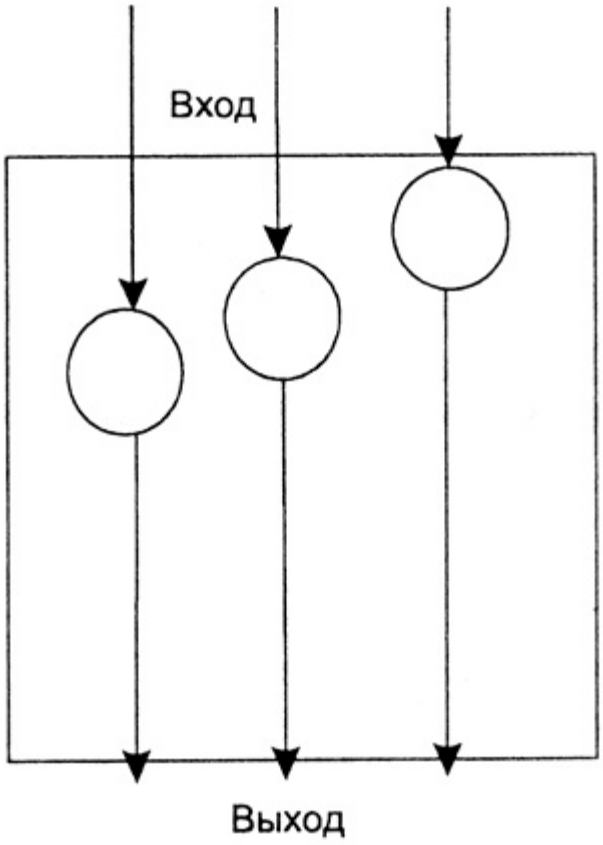


Рис. 2.3. *Полностью параллельный алгоритм*

Появление в середине шестидесятых годов прошлого столетия первого векторного

компьютера, разработанного в фирме CDC знаменитым Сеймуром Креем, ознаменовало рождение новой архитектуры. Основная идея, положенная в ее основу, заключалась в распараллеливании процесса обработки данных, когда одна и та же операция применяется одновременно к массиву (вектору) значений. В этом случае можно надеяться на определенный выигрыш в скорости вычислений. Идея параллелизма оказалась плодотворной и нашла воплощение на разных уровнях функционирования компьютера *(см. гл. 1).* В настоящее время, строго говоря, уже нет исключительно последовательных архитектур; параллелизм применяется на разных уровнях, а с параллелизмом на уровне команд приходится сталкиваться любому пользователю современного персонального компьютера.

Основными особенностями *модели параллельного программирования* являются более высокая производительность программ, применение специальных приемов программирования и, как следствие, более высокая трудоемкость программирования, проблемы с переносимостью программ. Параллельная модель не обладает свойством универсальности.

В параллельной модели программирования появляются проблемы, непривычные для программиста, привыкшего заниматься последовательным программированием. Среди них: управление работой множества процессоров, организация межпроцессорных пересылок данных и т. д. Можно сформулировать четыре фундаментальных требования к параллельным программам: параллелизм, масштабируемость, локальность и модульность.

В простейшей модели параллельного программирования, она называется *моделью задача/канал* (рис. 2.4), программа состоит из нескольких задач, связанных между собой каналами коммуникации и выполняющихся одновременно. Каждая задача состоит из последовательного кода и локальной памяти. Можно считать, что задача — это виртуальная фон-неймановская машина. Набор входных и выходных портов определяет ее интерфейс с окружением. Канал -- это очередь сообщений-посылок с данными. Задача может поместить в очередь свое сообщение или, наоборот, удалить сообщение, приняв содержащиеся в нем данные.

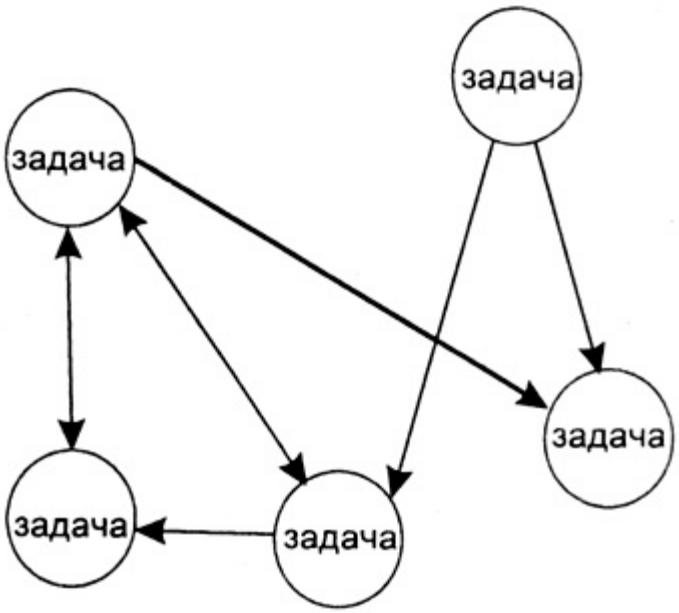


Рис. 2.4. *Модель задача/канал*

Количество задач может меняться в процессе выполнения. Кроме считывания и записи данных в локальную память каждая задача может посылать и принимать сообщения, порождать новые задачи и завершать их выполнение.

Операция отправки данных *асинхронная,* она завершается сразу. Операция приема *синхронная* — она блокирует выполнение задачи до тех пор, пока не будет получено сообщение. Количество каналов также может меняться в процессе выполнения программы.

Эффективность таких абстракций последовательного программирования, как процедуры и структуры данных, основана на том, что они легко отображаются на фон-неймановскую архитектуру. Задача и канал также сравнительно легко отображаются на архитектуру параллельной вычислительной системы. Если две задачи, разделяющие один канал, отображаются на разные процессоры, канал реализуется с помощью соответствующих механизмов межпроцессорных коммуникаций (это могут быть, например, сетевые протоколы). При отображении обеих задач на один процессор используются более эффективные способы коммуникации.

Независимость результата выполнения параллельной программы от отображения на конкретную архитектуру обеспечивается тем, что задачи взаимодействуют между собой с помощью универсального механизма. Он не зависит от того, как распределяются задачи, поэтому и результат выполнения программы не должен зависеть от способа распределения. При разработке алгоритмов в рамках параллельной -модели программирования не надо заботиться о количестве процессоров, на которых будет выполняться программа. Это позволяет добиться хорошей масштабируемости. При увеличении числа процессоров количество задач, приходящихся на один процессор, уменьшается, но алгоритм при этом изменять не

надо. Создание большего числа задач, чем число процессоров, может снизить потери быстродействия вследствие задержек при пересылке данных, поскольку во время пересылок и приема данных[7] одними задачами другие могут выполнять вычисления.

В модели задача/канал сохраняются и достоинства *модульного программирования.* Части модульной программы создаются отдельно и объединяются на завершающем этапе разработки программы. Взаимодействие между модулями обеспечивается хорошо определенными интерфейсами, а сами модули могут модифицироваться по отдельности, независимо от других модулей. Задача является естественным строительным блоком в модульной конструкции программы.

Есть сходство между данной моделью и объектно-ориентированной парадигмой программирования. Задачи, подобно объектам, содержат как данные, так и методы их обработки. Отличительные особенности модели задача/канал — параллельное выполнение, использование каналов, а не методов для определения взаимодействий и отсутствие поддержки наследования.

Детерминизм модели заключается в том, что результат выполнения программы с одним и тем же набором входных данных всегда дает один и тот же результат. Это свойство гарантируется тем, что каналу соответствует один отправитель, а при получении данных задачей ее выполнение блокируется.

Другие модели параллельного программирования

Существуют и другие модели параллельного программирования. Они различаются гибкостью, механизмами взаимодействия между задачами, уровнем параллелизма и "зернистостью", масштабируемостью, поддержкой модульности и т. д.

Передача сообщений

Передача сообщений — одна из самых распространенных моделей параллельного программирования. Программы, написанные в рамках этой модели, как и программы в модели задача/канал, при выполнении порождают несколько задач. Каждой задаче присваивается свой уникальный идентификатор, а взаимодействие осуществляется посредством отправки и приема сообщений. Можно считать, что данная модель программирования отличается от модели задача/канал только механизмом передачи данных. Сообщение посылается не в канал, а определенной задаче.

В модели передачи сообщений новые задачи могут создаваться во время выполнения параллельной программы, несколько задач могут выполняться на одном процессоре. Однако на практике при запуске программы чаще всего создается фиксированное число одинаковых задач, и это число остается неизменным во время выполнения программы. Такая разновидность модели называется *SPMD-моделью* (Single Program Multiple Data — одна программа, массив данных), поскольку каждая задача содержит

один и тот же код, но обрабатывает разные данные.

Параллелизм данных

Эта модель программирования основана на применении одной операции к множеству элементов структуры данных (пример такой операции - "увеличить в два раза стипендию всем студентам группы 111"). Программа, написанная в рамках данной модели, содержит последовательность таких операций. "Зернистость" вычислений мала, поскольку каждая операция над каждым элементом данных может считаться независимой задачей. Программист должен указать транслятору, как данные следует распределить между процессорами (т. е. между задачами). Транслятор генерирует SPMD-код, автоматически добавляя в него команды обмена данными. Методы разработки алгоритмов и анализа программ в модели с параллелизмом данных аналогичны тем, которые используются в модели задача/канал.

Модель разделяемой памяти

В модели разделяемой памяти задачи обращаются к общей памяти, имея общее адресное пространство и выполняя операции считывания/записи. Управление доступом к памяти осуществляется с помощью разных механизмов, таких, например, как семафоры. В рамках этой модели не требуется описывать обмен данными между задачами в явном виде. Это упрощает программирование. Вместе с тем особое внимание приходится уделять соблюдению детерминизма.

Закон Амдала

В общем случае структура информационного графа алгоритма занимает промежуточное положение между крайними случаями полностью последовательного и полностью параллельного алгоритма. В этой структуре имеются фрагменты, которые допускают одновременное выполнение на нескольких функциональных устройствах -- это параллельная часть программы. Есть и фрагменты, которые должны выполняться последовательно и на одном устройстве — это последовательная часть программы.

С помощью информационного графа можно оценить максимальное ускорение, которого можно достичь при распараллеливании алгоритма там, где это возможно. Предположим, что программа выполняется на машине, архитектура которой идеально соответствует структуре информационного графа программы. Пусть время выполнения алгоритма на последовательной машине T_1, причем Ts — время выполнения последовательной части алгоритма, а Tp — параллельной. Очевидно:

$$T_1 = T_S + T_p.$$

При выполнении той же программы на идеальной параллельной машине, N независимых ветвей параллельной части распределяются по одной на V процессоров,

поэтому время выполнения этой части уменьшается до величины *Tp / N, а* полное время выполнения программы составит:

$T_2=T_S+ Tp/N.$

Коэффициент ускорения, показывающий, во сколько раз быстрее программа выполняется на параллельной машине, чем на последовательной, определяется формулой:

$X= T_1/T_2 = (T_s + Tp) / (T_s + T_p/N) = 1/(S + P/N),$

где $S= T_s / (T_s + T_p)$ и $P= Tp / (T_s + T_p)$ — относительные доли последовательной и параллельной частей $(S+ P= 1)$. График зависимости коэффициента ускорения от числа процессоров и степени параллелизма алгоритма (относительной доли параллельной части) приведен на рис. 2.5. Эта зависимость носит название *закона Амдала.*

Из рисунка видно, что для программ (алгоритмов) с небольшой степенью параллелизма использование большого числа процессоров не дает сколько-нибудь значительного выигрыша в быстродействии. Если же степень параллелизма достаточно велика, коэффициент ускорения может быть большим. Начиная с некоторого значения, увеличение числа процессоров дает только небольшой прирост производительности. На практике, когда приходится принимать во внимание конечное время обмена данными, при увеличении числа процессоров может наблюдаться даже падение производительности, поскольку увеличивается количество обменов.

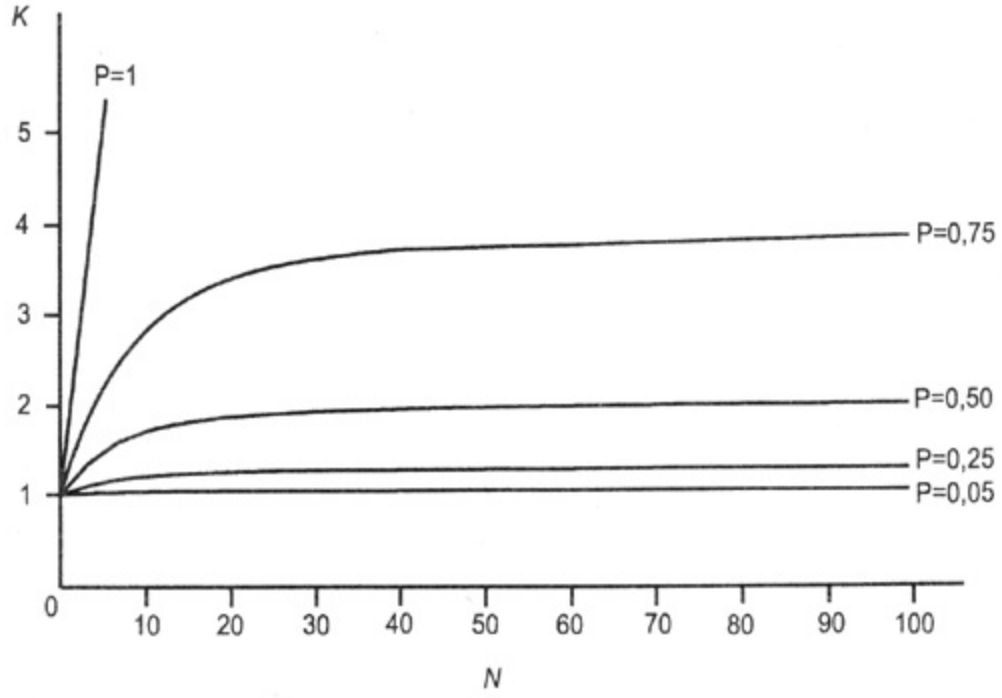


Рис. 2.5. *Закон Амдала*

Приведем различные количественные характеристики параллелизма. *Эффективность использования процессоров* характеризуется величиной:

E=K/N, 1/N<=E<=1.

В идеальной ситуации *K= N и. E= 1,* но на практике это значение эффективности недостижимо.

Степенью параллелизма называют количество процессоров, используемых в каждый момент времени для выполнения программы. Степень параллелизма может изменяться в процессе выполнения программы. Это, вообще говоря, переменная величина, которая зависит, в частности, от наличия и доступности ресурсов, поэтому иногда вводят различные средние характеристики параллелизма.

Исследования показали, что потенциальный параллелизм в научных и технических прикладных программах может быть очень большим, до сотен и тысяч команд на такт, однако *реальный* ("достижимый") параллелизм значительно меньше — 10—20.

Две парадигмы параллельного программирования

Наиболее распространенными подходами к распараллеливанию вычислений и обработки данных являются подходы, основанные на моделях параллелизма данных и параллелизма задач. В основе каждого подхода лежит распределение вычислительной работы между доступными пользователю процессорами параллельного компьютера. При этом приходится решать разнообразные проблемы. Прежде всего, это равномерная загрузка процессоров, т. к. если основная вычислительная работа будет ложиться только на часть из них, уменьшится и выигрыш от распараллеливания. Другая не менее важная проблема — эффективная организация обмена информацией между задачами. Если вычисления выполняются на высокопроизводительных процессорах, загрузка которых достаточно равномерна, но скорость обмена данными при этом низка, большая часть времени будет тратиться впустую на ожидание информации, необходимой для дальнейшей работы задачи. Это может существенно снизить скорость работы программы.

Параллелизм данных

Мы уже упоминали о том, что основная идея подхода, основанного на параллелизме данных, — применение одной операции сразу к нескольким элементам массива данных. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Векторизация или распараллеливание в этом случае чаще всего выполняются уже во время трансляции - перевода исходного текста программы в машинные команды. Роль программиста в этом случае обычно сводится к заданию опций векторной или параллельной оптимизации транслятору, директив параллельной компиляции, использованию

специализированных языков параллельных вычислений.

Основные особенности рассматриваемого подхода: П обработкой данных управляет одна программа;

- пространство имен является глобальным, т. е. для программиста существует одна единственная память, а детали структуры данных, доступа к памяти и межпроцессорного обмена данными от него скрыты;
- слабая синхронизация вычислений на параллельных процессорах, т. е. выполнение команд на разных процессорах происходит, как правило, независимо и только иногда производится согласование выполнения циклов или других программных конструкций — их синхронизация. Каждый процессор выполняет один и тот же фрагмент программы, но нет гарантии, что в заданный момент времени на всех процессорах выполняется одна и та же машинная команда;
- параллельные операции над элементами массива выполняются одновременно на всех доступных данной программе процессорах.

Видим, таким образом, что в рамках данного подхода от программиста не требуется больших усилий по векторизации или распараллеливанию вычислений. Даже при программировании сложных вычислительных алгоритмов можно использовать библиотеки подпрограмм, специально разработанных с учетом конкретной архитектуры компьютера и оптимизированных для этой архитектуры.

Подход, основанный на параллелизме данных, базируется на использовании в программах базового набора операций:

- операции управления данными;
- операции над массивами в целом, а также их фрагментами;
- условные операции;
- операции приведения;
- операции сдвига;
- операции сканирования;
- операции, связанные с пересылкой данных.

Рассмотрим эти базовые операции.

Управление данными

В определенных ситуациях возникает необходимость в управлении распределением данных между процессорами. Это может потребоваться, например, для обеспечения равномерной загрузки процессоров. Чем более равномерно загружены работой процессоры, тем более эффективной будет работа компьютера.

Операции над массивами

Аргументами этих операций являются массивы в целом или их фрагменты (сечения), при этом одна операция применяется одновременно ко всем элементам массива или его части. Примером операции такого типа является умножение элементов массива на скалярный или векторный множитель. Операции могут быть и более сложными — вычисление функции от массива, например.

Условные операции

Условные операции применяются лишь к тем элементам массива, которые удовлетворяют определенному условию. В сеточных методах, например, это могут быть элементы, соответствующие четным/нечетным номерам строк или столбцов сетки.

Операции приведения

Операции приведения применяются ко всем элементам массива (или его части), а результатом является одно-единственное значение. Примерами операции приведения являются операции вычисления суммы элементов массива или максимального значения его элементов.

Операции сдвига

Для эффективной реализации некоторых параллельных алгоритмов требуются операции сдвига массивов. Сдвиги используются в некоторых алгоритмах обработки изображений, в конечно-разностных алгоритмах и т. д.

Операции сканирования

Операции сканирования называются также *префиксными/суффиксными* операциями. Префиксная операция суммирования, например, выполняется следующим образом. Элементы массива суммируются последовательно, а результат очередного суммирования заносится в очередную ячейку нового, результирующего массива, причем номер этой ячейки совпадает с числом просуммированных элементов исходного массива.

Операции пересылки данных

Операции пересылки данных могут осуществляться, например, между массивами различной формы, имеющими разную размерность, протяженность по каждому измерению и пр.

Программирование в модели параллелизма данных

При программировании на основе параллелизма данных часто используются специализированные языки - CM FORTRAN, C*, FORTRAN+, MPP FORTRAN, Vienna

FORTRAN, а также HPF (High Perfomance FORTRAN). Язык HPF основан на языке программирования FORTRAN-90, что связано с наличием в нем удобных операций над массивами.

Реализация модели параллелизма данных требует поддержки параллелизма на уровне транслятора. Такую поддержку могут обеспечивать:

- *препроцессоры,* использующие существующие последовательные трансляторы и специализированные библиотеки, с реализациями параллельных алгоритмических конструкций;
- *предтрансляторы,* которые выполняют предварительный анализ логической структуры программы, проверку зависимостей и ограниченную параллельную оптимизацию;
- *распараллеливающие трансляторы,* которые выявляют параллелизм в исходном коде программы и выполняют его преобразование в параллельные конструкции. Для того чтобы упростить преобразование, в исходный текст программы могут добавляться специальные директивы трансляции.

Параллелизм задач

Метод программирования, основанный на параллелизме задач, предусматривает разбиение вычислительной задачи на несколько относительно самостоятельных подзадач. Каждая подзадача выполняется на своем процессоре. Компьютер при этом представляет собой MIMD-машину. Для каждой подзадачи пишется своя собственная программа на обычном языке программирования, чаще всего это FORTRAN или C. Подзадачи должны обмениваться результатами своей работы, получать исходные данные. Практически такой обмен осуществляется вызовом процедур специализированной библиотеки. Программист при этом может контролировать распределение данных между различными процессорами и различными подзадачами, а также обмен данными. В этом случае, очевидно, требуется дополнительная работа для того, чтобы обеспечить эффективное совместное выполнение различных подзадач. По сравнению с подходом, основанным на параллелизме данных, этот подход более трудоемкий, с ним связаны следующие проблемы:

- повышенная трудоемкость как разработки программы, так и ее отладки;
- на программиста ложится вся ответственность за равномерную и сбалансированную загрузку процессоров параллельного компьютера;
- программисту приходится минимизировать обмен данными между задачами, т. к. затраты времени на пересылку данных обычно относительно велики;
- повышенная опасность возникновения тупиковых ситуаций, когда отправленная одной программой посылка с данными не приходит к месту назначения.

Привлекательными особенностями данного подхода являются большая гибкость и

большая свобода, предоставляемая программисту в разработке программы, эффективно использующей ресурсы параллельного компьютера и, как следствие, возможность достижения максимального быстродействия. Примерами специализированных библиотек являются библиотеки MPI (Message Passing Interface) и PVM (Parallel Virtual Machines). Эти библиотеки распространяются свободно и существуют в исходных кодах. Библиотека MPI разработана в Аргоннской .Национальной Лаборатории (США), а PVM — разработка Окриджской Национальной Лаборатории, университетов штата Теннесси и Эмори (США).

Разработка параллельного алгоритма

Важнейшим и одним из наиболее трудоемких этапов создания программы является разработка алгоритма. Если же речь идет о разработке параллельного алгоритма, возникают дополнительные проблемы, о которых уже упоминалось ранее в этой главе. Алгоритм должен обеспечивать эффективное использование параллельной вычислительной системы. Следует заботиться и о хорошей масштабируемости алгоритма, а это не всегда просто.

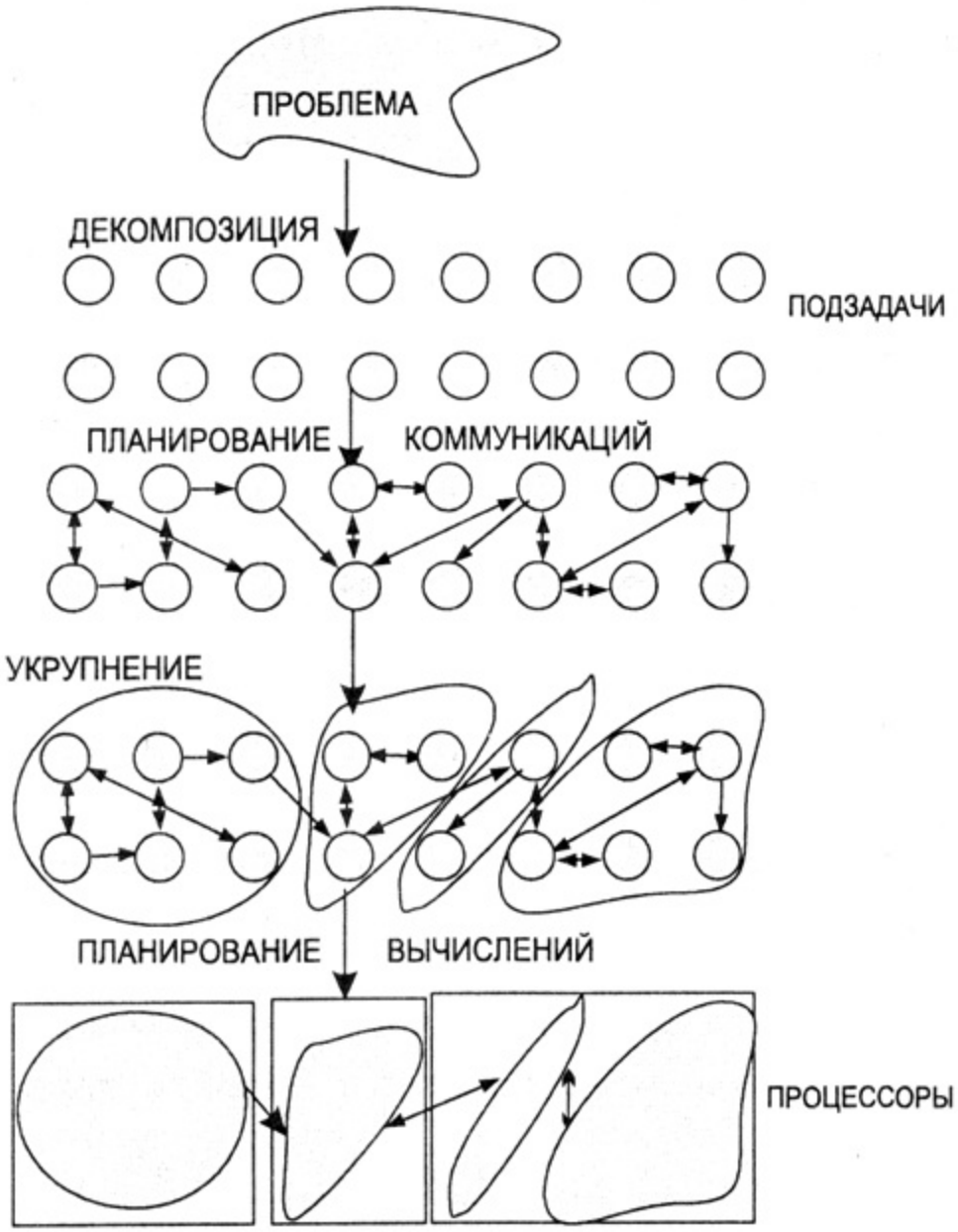


Рис. 2.6. *Этапы разработки параллельного алгоритма*

Процесс разработки параллельного алгоритма можно разбить на четыре этапа (рис. 2.6).

1. *Декомпозиция.* На данном этапе исходная задача анализируется, оценивается возможность ее распараллеливания. Иногда выигрыш от распараллеливания может быть незначительным, а трудоемкость разработки параллельной программы большой. В этом случае первый этап разработки алгоритма оказывается и последним. Если же "игра стоит свеч", задача и связанные с ней данные разделяются на более мелкие части — подзадачи и фрагменты структур данных. Особенности архитектуры конкретной вычислительной системы на данном этапе могут не учитываться.

2. *Проектирование коммуникаций (обменов данными) между задачами.* На этом этапе определяются коммуникации, необходимые для пересылки исходных данных, промежуточных результатов выполнения подзадач, а также коммуникации,

необходимые для управления работой подзадач. Выбираются алгоритмы и методы коммуникации.

3. *Укрупнение.* Подзадачи могут объединяться в более крупные блоки, если это позволяет повысить эффективность алгоритма и снизить трудоемкость разработки. Основными критериями на данном этапе являются эффективность алгоритма (производительность — в первую очередь) и трудоемкость его реализации.

4. *Планирование вычислений.* На этом заключительном этапе производится распределение подзадач между процессорами. Основной критерий выбора способа размещения подзадач — эффективное использование процессоров с минимальными затратами времени на обмены данными.

Рассмотрим каждый из этих этапов более подробно.

Декомпозиция (сегментирование)

На данном этапе определяется степень возможного распараллеливания задачи. Иногда декомпозиция исходной задачи естественным образом следует из природы задачи и оказывается очевидной. Иногда это не так. Чем меньше размер подзадач, получаемых в результате декомпозиции, чем больше их количество, тем более гибким оказывается параллельный алгоритм, тем легче обеспечить равномерную загрузку процессоров вычислительной системы. Впоследствии, возможно, придется "укрупнить" алгоритм, поскольку следует принять во внимание интенсивность обмена данными и другие факторы.

Сегментироваться могут как вычислительный алгоритм, так и данные. Применяют разные варианты декомпозиции. В методе *декомпозиции данных* сначала сегментируются данные, а затем алгоритм их обработки. Данные разбиваются на фрагменты приблизительно одинакового размера. С фрагментами данных связываются операции их обработки, из которых и формируются подзадачи. Определяются необходимые пересылки данных. Пересечение частей, на которые разбивается задача, должно быть сведено к минимуму. Это позволяет избежать дублирования вычислений. Схема разбиения в дальнейшем может уточняться. В частности, если это необходимо для уменьшения числа обменов, допускается увеличение степени перекрытия фрагментов задачи.

При декомпозиции данных сначала анализируются структуры данных наибольшего размера, либо те, обращение к которым происходит чаще всего. На разных стадиях расчета могут использоваться различные структуры данных, поэтому могут потребоваться и динамические, т. е. изменяющиеся со временем, схемы декомпозиции этих структур.

Приведем пример. В инженерных и научных расчетах часто используются различные

сеточные методы. В этом случае трехмерная сетка определяет набор данных, элементы которого сопоставляются узлам решетки. Декомпозиция в данном случае может быть одномерной (это самое крупноблочное разбиение), двумерной и трехмерной (рис. 2.7). В последнем случае отдельными фрагментами являются узлы сетки.

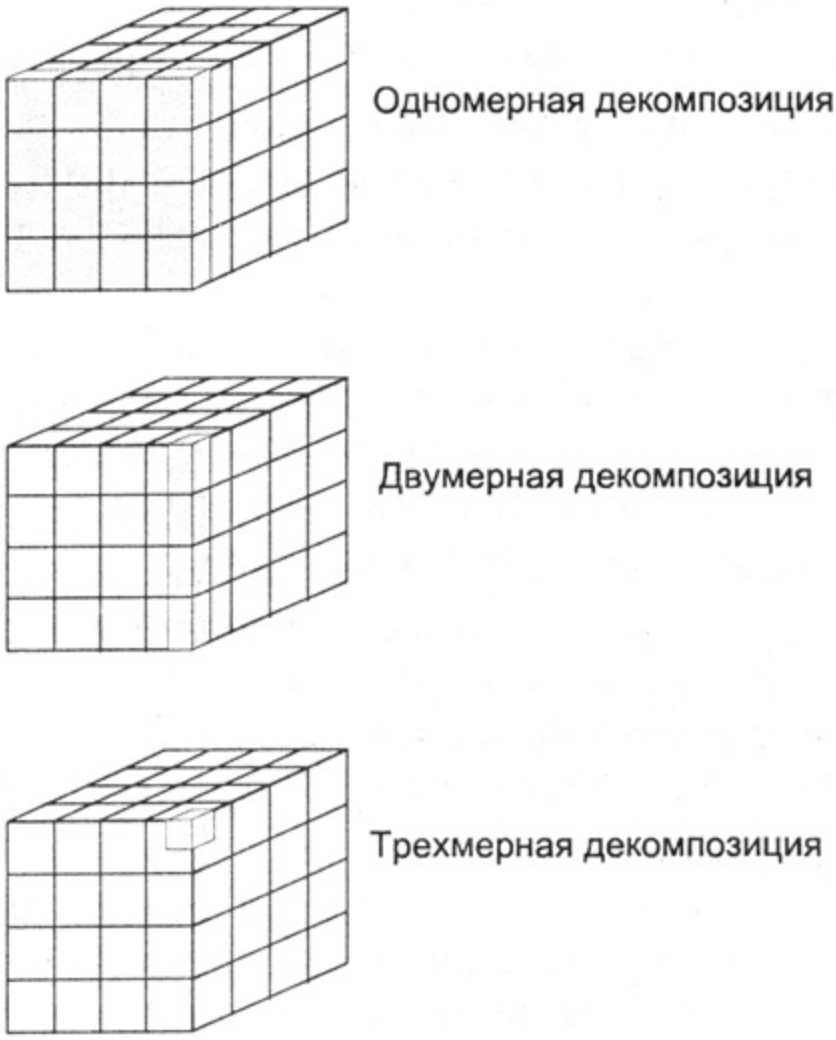


Рис. 2.7. *Декомпозиция трехмерной сетки*

В методе *функциональной декомпозиции* сначала сегментируется вычислительный алгоритм, а затем уже под эту схему подгоняется схема декомпозиции данных. Успех при этом не всегда гарантирован. Схема может оказаться такой, что потребуются многочисленные дополнительные пересылки данных. Метод функциональной декомпозиции может оказаться полезным в ситуации, где нет структур данных, которые очевидно могли бы быть распараллелены.

Функциональная декомпозиция играет важную роль и как метод структурирования программ. Он может оказаться полезным, например, при моделировании сложных систем, которые состоят из множества простых подсистем, связанных между собой набором интерфейсов.

На практике чаще используется метод декомпозиции данных. Для того чтобы декомпозиция была эффективной, следует придерживаться следующих рекомендаций:

- количество подзадач после декомпозиции должно примерно на порядок превосходить количество процессоров. *Это* позволяет обеспечить большую гибкость на последующих этапах разработки программы;
- следует избегать лишних вычислений и пересылок данных, в противном случае программа может оказаться плохо масштабируемой и не позволит достичь высокой эффективности при решении задач большого объема;
- подзадачи должны быть примерно одинакового размера, в этом случае легче обеспечить сбалансированную загрузку процессоров;
- в идеале сегментация должна быть такой, чтобы с увеличением объема задачи количество подзадач также возрастало (при сохранении постоянным размера одной подзадачи). Это обеспечит хорошую масштабируемость.

Размер блоков, из которых состоит параллельная программа, может быть разным. В зависимости от размера блоков алгоритм может иметь различную "зернистость". Ее мерой в простейшем случае может служить количество операций в блоке. Выделяют три степени "зернистости": мелкозернистый, среднеблочный и крупноблочный параллелизм.

Зернистость связана с уровнем параллелизма. Параллелизм на уровне команд — самый "мелкозернистый". Его масштаб менее 20 команд на блок. Количество параллельно выполняемых подзадач — от единиц до нескольких тысяч, причем средний масштаб параллелизма составляет около 5 команд на блок.

Следующий уровень — это параллелизм на уровне циклов. Обычно цикл содержит не более 500 команд. Если итерации цикла независимы, они могут выполняться, например, с помощью конвейера или на векторном процессоре. Это тоже "мелкозернистый" параллелизм.

Параллелизм на уровне процедур — среднеблочный. Размер блока до 2000 команд. Выявление такого параллелизма сложнее реализовать, поскольку следует учитывать возможные межпроцедурные зависимости. Требования к коммуникациям меньше, чем в случае параллелизма на уровне команд.

Параллелизм на уровне программ (задач) — это уже крупноблочный параллелизм. Он соответствует выполнению независимых программ на параллельном компьютере. Крупноблочный параллелизм требует поддержки операционной системой.

Обмен данными через разделяемые переменные используется на уровне мелкозернистого и среднеблочного параллелизма, а посредством сообщений — на среднем и крупном уровнях.

Добиться эффективной работы параллельной программы можно, сбалансировав "зернистость" алгоритма и затраты времени на обмен данными. Различные подсистемы параллельного компьютера характеризуются разными временами задержки. Например, время доступа к памяти увеличивается с ростом ее объема. Время задержки определяет степень масштабируемости системы.

Части программы могут выполняться параллельно, только если они независимы. *Независимость по данным* заключается в том, что данные, обрабатываемые одной частью программы, не модифицируются другой ее частью. *Независимость по управлению* состоит в том, что порядок выполнения частей программы может быть определен только во время выполнения программы (наличие зависимости по управлению предопределяет последовательность выполнения). *Независимость по ресурсам* обеспечивается достаточным количеством компьютерных ресурсов (объемом памяти, количеством функциональных узлов и т. д.). *Зависимость по выводу* возникает, если две подзадачи производят запись в одну и ту же переменную, а зависимость по вводу/выводу, если операторы ввода/вывода двух или нескольких подзадач обращаются к одному файлу (или переменной). В листингах 2.1 и 2.2 приведены фрагменты программы на языке FORTRAN, независимые и зависимые по управлению соответственно.

Две подзадачи могут выполняться параллельно, если они независимы по данным, по управлению и по операциям вывода.

Листинг 2.1. Фрагмент кода, демонстрирующий независимость по управлению

```
do k = 1, m

a[k] = b[k]

if (a[k] < c) then a[k] = 1

endif enddo
```

Листинг 2.2. Фрагмент кода, демонстрирующий зависимость по управлению

```
do k = 1, m

if (a[k - 1] < c) then a[k] = 1

endif enddo
```

Свойство параллельности обладает свойством коммутативности — если подзадачи *a* и *b* параллельны, *то b и a* тоже параллельны, но не транзитивно, т. е. если *a* и *b*

параллельны, а также параллельны *b* и *c,* отсюда не обязательно следует параллельность *a и c.*

Декомпозиция должна быть такой, чтобы время счета или обработки данных превосходило время, затрачиваемое на пересылку данных. Вряд ли существуют точные алгоритмы такой декомпозиции, можно считать, что сегментирование алгоритма — это искусство.

Проектирование коммуникаций

Подзадачи не могут быть абсолютно независимыми, поэтому следующим этапом разработки алгоритма является проектирование обменов данными между ними. Проектирование заключается в определении структуры каналов связи и сообщений, которыми должны обмениваться подзадачи. Каналы связи могут создаваться неявно, как это происходит, например, в языках программирования, поддерживающих модель параллелизма данных, а могут программироваться явно.

Если на первом этапе применяется декомпозиция данных, проектирование коммуникаций может оказаться непростым делом. Сегментация данных на непересекающиеся подмножества и связывание с каждым из них своего множества операций обработки — достаточно очевидный шаг. Однако остается необходимость обмена данными, а также управления этим обменом. Структура коммуникаций может оказаться сложной.

В схеме функциональной декомпозиции организация коммуникаций обычно проще — они соответствуют потокам данных между задачами.

Коммуникации бывают следующих типов:

- *локальные* — каждая подзадача связана с небольшим набором других подзадач;
- *глобальные* — каждая подзадача связана с большим числом других подзадач;
- *структурированные* — каждая подзадача и подзадачи, связанные с ней, образуют регулярную структуру (например, с топологией решетки);
- *неструктурированные* — подзадачи связаны произвольным графом;
- *статические* — схема коммуникаций не изменяется с течением времени;
- *динамические* — схема коммуникаций изменяется в процессе выполнения программы;
- *синхронные* — отправитель и получатель данных координируют обмен;
- *асинхронные* — обмен данными не координируется.

В глобальной схеме обменов может быть слишком много, что ухудшает масштабируемость программы. Примером использования такой схемы являются программы, организованные по принципу хозяин/работник (master/slave), когда один процесс запускает множество подзадач, распределяет между ними работу и

принимает результаты. Недостатком программы с такой организацией может быть то, что главная программа принимает сообщения от подчиненных задач по очереди. В этом случае она работает в последовательном режиме.

Неструктурированные коммуникации применяются, например, в методах конечных элементов, где приходится иметь дело с геометрическими объектами сложной формы, а также при использовании в сеточных методах неравномерных сеток. Структура сетки может изменяться в процессе расчета. На первых этапах разработки параллельного алгоритма неструктурированный характер коммуникаций обычно не доставляет трудностей. Проблемы могут возникать на этапах укрупнения и планирования вычислений.

В случае асинхронных коммуникаций задачи-отправители данных не могут определить, когда данные требуются другим задачам, поэтому последние должны сами запрашивать данные.

Вот несколько рекомендаций по проектированию коммуникаций:

- количество коммуникаций у подзадач должно быть примерно одинаковым, иначе приложение становится плохо масштабируемым;
- там, где это возможно, следует использовать локальные коммуникации;
- коммуникации должны быть, по возможности, параллельными.

Оптимальная организация обмена данными между подзадачами должна учитывать архитектуру коммуникационной сети высокопроизводительной вычислительной системы, должна обеспечивать ее равномерную загрузку и минимизировать конфликты по доступу к различным узлам системы.

Тупиковые ситуации могут быть связаны с неправильной последовательностью обмена данными между процессами, когда, например, первый процесс ожидает данные от второго, тот, в свою очередь, от третьего, а третьему процессу требуется результат работы первого процесса. Решить проблему тупиков можно, используя неблокирующий прием данных, сочетая прием данных с их отправкой и т. д.

Обмен сообщениями может быть реализован по-разному: с помощью потоков, межпроцессных коммуникаций (IPC - Inter-Process Communication), TCP-сокетов и т. д. Один из самых распространенных способов программирования коммуникаций -- использование библиотек, реализующих обмен сообщениями. Например, уже упоминавшиеся библиотеки PVM (Parallel Virtual Machine) и MPI (Message Passing Interface), которые позволяют создавать параллельные программы для самых разных платформ. Так, программы, написанные для кластера, могут выполняться на суперкомпьютере.

Существуют и другие способы организации коммуникаций. Метод RPC (Remote

Procedure Control — удаленное управление процедурами) впервые был предложен фирмой Sun Microsystems. Он позволяет одному процессу вызывать процедуру из другого процесса, передавать ей параметры и, при необходимости, получать результаты выполнения.

CORBA (Common Object Request Broker Architecture) определяет протокол взаимодействия между процессами, независимый от языка программирования и операционной системы. Для описания интерфейсов используется декларативный язык IDL (Interface Definition Language).

DCOM (Distributed Component Object Model, разработка Microsoft) дает приблизительно такие же возможности, что и CORBA. Лежит в основе интер-

фейсов ActiveX, с помощью которых одно приложение MS Windows может запустить другое приложение и управлять его выполнением.

Укрупнение

Результатом первых двух этапов разработки параллельного алгоритма является алгоритм, который не ориентирован на конкретную архитектуру вычислительной системы, поэтому он может оказаться неэффективным. На этапе укрупнения учитывается архитектура вычислительной системы, при этом часто приходится объединять (укрупнять) задачи, полученные на первых двух этапах, для того чтобы их число соответствовало числу процессоров.

Пример укрупнения приведен на рис. 2.8.

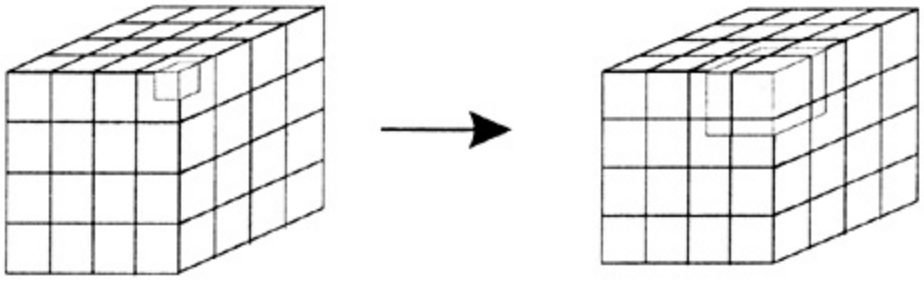


Рис. 2.8. *Укрупнение в трехмерном сеточном методе*

При укрупнении могут преследоваться такие цели, как увеличение "зернистости" вычислений и коммуникаций, сохранение гибкости и снижение стоимости разработки. Основные требования:

- снижение затрат на коммуникации;
- если при укрупнении приходится дублировать вычисления или данные, это не должно приводить к потере производительности и масштабируемости программы;
- результирующие задачи должны иметь примерно одинаковую трудоемкость;

- сохранение масштабируемости;
- сохранение возможности параллельного выполнения;
- снижение стоимости и трудоемкости разработки.

Планирование вычислений

На этапе планирования вычислений, заключительном этапе разработки параллельного алгоритма, необходимо определить, где, на каких процессорах будут выполняться подзадачи. Основным критерием эффективности здесь является минимизация времени выполнения программы.

Проблема планирования легко решается при использовании систем с разделяемой памятью, однако не всегда удается найти эффективный способ укрупнения и планирования вычислений. В этом случае применяют эвристические алгоритмы динамически сбалансированной загрузки процессоров, когда переопределение стратегии укрупнения и планирования производится периодически во время выполнения программы и иногда с достаточно большой частотой.

В алгоритмах, основанных на функциональной декомпозиции, часто создается множество мелких "короткоживущих" подзадач. В этом случае применяются *алгоритмы планирования выполнения задач* (tasks scheduling), которые распределяют задачи по не загруженным работой процессорам.

Планирование выполнения задач заключается в организации их доступа к ресурсам. Порядок предоставления доступа определяется используемым для этого алгоритмом. В качестве примера можно привести *планирование по принципу кругового обслуживания* (round robin). Это алгоритм планирования, когда несколько процессов обслуживаются по очереди, получая одинаковые порции процессорного времени. Часто используется также *метод сбалансированной загрузки* (load balancing) процессоров вычислительной системы. Он основан на учете текущей загрузки каждого процессора. Иногда при реализации метода сбалансированной загрузки предусматривается перенос ("миграция") процесса с одного процессора на другой.

В алгоритмах *планирования выполнения задач* формируется набор (пул) задач, в который включаются вновь созданные задачи и из которого задачи выбираются для выполнения на освободившихся процессорах. Стратегия размещения задач на процессорах обычно представляет собой компромисс между требованием максимальной независимости выполняющихся задач (минимизация коммуникаций) и глобальным учетом состояния вычислений. Чаще всего применяются стратегии хозяин/работник (рис. 2.9), иерархические и децентрализованные стратегии.

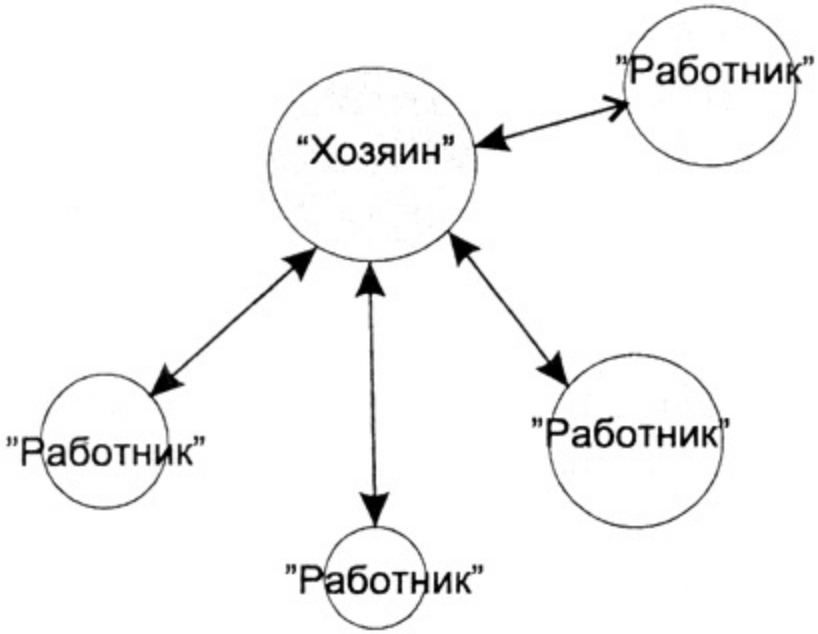


Рис. 2.9. *Стратегия управления хозяин/работник*

В этой простой в реализации схеме (эффективной для небольшого числа процессоров) главная задача отвечает за размещение подчиненных задач. Подчиненная задача получает исходные данные для обработки от главной задачи и возвращает ей результат работы. Эффективность данной схемы зависит от числа подчиненных задач и относительных затрат на обмены данными между подчиненной и главной задачами. При использовании централизованной схемы сбалансированной загрузки главная задача не должна быть "слабым звеном" программы, т. е. она не должна тормозить выполнение других задач.

Иерархическая схема хозяин/работник является разновидностью простой схемы хозяин/работник, в которой подчиненные задачи разделены на непересекающиеся подмножества и у каждого из этих подмножеств есть своя главная задача. Главные задачи подмножеств управляются одной "самой главной" задачей.

В децентрализованных схемах главная задача отсутствует. Задачи могут обмениваться данными друг с другом, придерживаясь определенной стратегии. Это может быть случайный выбор объекта коммуникации или взаимодействие с небольшим числом ближайших соседей. В гибридной централизованно/распределенной схеме запрос посылается главной задаче, а она передает его подчиненным задачам, используя метод кругового планирования.

При решении сложных задач динамические методы планирования проще, но производительность программы при этом может быть меньше. С другой стороны, алгоритм SPMD обеспечивает более полный контроль над коммуникациями и вычислениями, хотя он и сложнее.

Динамически сбалансированная загрузка может быть эффективно реализована, если учтены следующие соображения:

- если каждый процессор выполняет одну подзадачу, длительность выполнения всей программы будет определяться временем выполнения самой длительной подзадачи;
- оптимальная производительность достигается, если все подзадачи имеют примерно одинаковый размер;
- заботиться о сбалансированной работе процессоров может программист, но имеются и средства автоматической поддержки сбалансированной работы;
- сбалансированность может быть обеспечена посредством загрузки каждого процессора несколькими задачами.

Существуют различные алгоритмы сбалансированной загрузки, применяемые в методах декомпозиции данных. Это методы рекурсивной дихотомии, локальные алгоритмы, вероятностные методы, циклические отображения и т. д. Все они предназначены для укрупнения мелкозернистых задач, так, чтобы в результате на один процессор приходилась одна крупноблочная задача.

Будем считать, что данные связаны с определенной пространственной структурой, например, трехмерной сеткой, и эту структуру будем называть *областью. Рекурсивная дихотомия* используется для разбиения области на подобласти, которым соответствует примерно одинаковая трудоемкость вычислений, а коммуникации сведены к минимуму. Область сначала разбивается на две части вдоль одного измерения. Разбиение повторяется рекурсивно в каждой новой подобласти столько раз, сколько потребуется для получения необходимого числа подзадач.

Метод *рекурсивной координатной дихотомии* обычно применяется к нерегулярным сеткам. Деление выполняется на каждом шаге вдоль того измерения, по которому сетка имеет наибольшую протяженность. Это достаточно простой метод, который, однако, не позволяет добиться большой эффективности коммуникаций. Здесь возможны ситуации, когда возникают протяженные подобласти с большим числом локальных коммуникаций.

Метод рекурсивной дихотомии графа используется для нерегулярных сеток. В нем с помощью информации о топологии решетки минимизируется количество ребер, пересекающих границы подобластей. Это позволяет снизить количество коммуникаций.

Локальные алгоритмы, обеспечивающие динамически сбалансированную загрузку процессоров, используют информацию только о ближайших соседях (подзадачах или процессорах). В отличие от глобальных алгоритмов они более экономны и чаще используются в тех ситуациях, когда загрузка быстро меняется со временем. Однако

эффективность локальных алгоритмов меньше, чем глобальных, поскольку они используют ограниченную информацию о состоянии вычислительного процесса.

Вероятностные методы планирования экономны и позволяют обеспечить хорошую масштабируемость приложений. В этом случае задачи размещаются для выполнения на случайно выбранных процессорах. Если количество задач велико, средняя загрузка всех процессоров будет одинаковой. Недостатком данного подхода можно считать то, что для равномерной загрузки количество задач должно намного превосходить количество процессоров. Наибольшую эффективность вероятностные методы показывают в ситуациях с небольшим числом обменов.

Циклическое планирование является разновидностью вероятностного планирования. В этом случае выбирается определенная схема нумерации подзадач, и каждый *N-й* процессор загружается каждой *N-й* задачей.

Количественные характеристики быстродействия

Важнейшим критерием эффективности параллельного программирования является быстродействие программы, ее производительность. На производительность влияют различные факторы. Это технология выполнения аппаратной части (в том числе электронных компонентов), архитектура вычислительной системы, методы управления ресурсами, эффективность параллельного алгоритма, особенности структуры данных, эффективность языка программирования, квалификация программиста, эффективность транслятора и т. д. Время выполнения программы зависит от времени доступа к главной и внешней памяти, количества операций ввода и вывода, загруженности операционной системы.

Процессор управляется тактовым генератором, вырабатывающим управляющие импульсы фиксированной длительности — такты. Обратная длительности импульса величина называется частотой. Выполнение каждой машинной команды требует нескольких тактов. Количество тактов на команду (CPI — Cycles Per Instruction) характеризует трудоемкость и длительность команды. В разных классах программ разное среднее значение CPI, поэтому оно может служить численной характеристикой программы.

Процессорное время, необходимое для выполнения программы, можно определить по формуле:

T=NiCPIt,

где *Ni*, — количество машинных команд в программе, *а t* — длительность такта.

Быстродействие процессора измеряется в MIPS (Million Instructions Per Second). MIPS обратно пропорционально CPI.

После того, как разработан алгоритм и проведен предварительный анализ быстродействия программы, а также проведены оценки трудоемкости и целесообразности разработки параллельной программы, наступает этап кодирования. Этому этапу, собственно, посвящена большая часть последующих глав.

Программные средства высокопроизводительных вычислений

Процесс разработки параллельной программы отличается от процесса разработки последовательной программы. В этом случае требуется предварительный анализ, позволяющий выяснить, какие фрагменты программы "съедают" наибольшее количество процессорного времени. Распараллеливание именно таких фрагментов позволяет добиться увеличения скорости выполнения программы. Распараллеливание связано с выявлением подзадач, решение которых можно поручить различным процессорам. Требуется организация взаимодействия между такими подзадачами, что делается путем обмена сообщениями-посылками, содержащими исходные данные, промежуточные и окончательные результаты работы. Специальные приемы требуются и при отладке параллельной программы.

Эффективность разработки программ зависит от наличия соответствующего программного инструментария. Разработчику параллельных программ требуются средства анализа и выявления параллелизма, трансляторы, операционные системы, обеспечивающие надежную работу многопроцессорных конфигураций. Система разработки параллельных программ должна включать в свой состав также средства отладки и профилирования (оценки производительности программы и отдельных ее частей). Средой выполнения параллельных программ являются операционные системы с поддержкой мультипроцессирования, многозадачности и многопоточности. Чаще всего это операционные системы семейства UNIX или Microsoft Windows NT/2000.

В табл. 2.1 приведен неполный список параллельных языков программирования и систем разработки параллельных программ.

Таблица 2.1. *Параллельные языки и системы программирования*

Для систем с разделяемой памятью	**Для систем с распределенной памятью**	**Параллельные объектно-ориентированные**	**Параллельные декларативные**
ОрепМР	PVM	HPC++	Parlog
Linda	MPI	MPL	Multilisp

Orca	HPF	CA	Sisal
Java	Cilk	Distributed Java	Concurrent Prolog
Pthreads	C*	Charm++	GHC
Opus	ZPL	Concurrent Aggregates	Strand
SDL	Occam	Argus	Tempo
Ease	Concurrent C	Presto	
SHMEM	Ada	Nexus	
	FORTRAN M	uC++	
	CSP	sC++	
	NESL	pC++	
	MpC		

Языки и системы параллельного программирования находятся в состоянии постоянного развития. Появляются новые концепции и идеи, совершенствуются ставшие уже традиционными системы параллельного программирования.

Некоторые языки параллельного программирования машинно-зависимы и созданы для конкретных вычислительных систем. Приложения, написанные на таких языках, непереносимы. Используются и универсальные языки программирования высокого уровня, такие как современные версии языка FORTRAN и язык C.

FORTRAN D - это расширения языков FORTRAN 77/90 (FORTRAN 77D/90D), с помощью которых программист может определить машинно-независимым образом, как следует распределить данные между процессорами. Язык позволяет использовать общее пространство имен. Трансляторы могут генерировать код как для SIMD, так и для MIMD-машин. Система D "выросла" из набора инструментальных

средств ParaScope, разработанного для поддержки создания параллельных программ с явным параллелизмом. Система D ориентирована на программирование для систем с распределен-но-разделяемой памятью.

FORTRAN D послужил предшественником языка High Performance FORTRAN (HPF). Спецификация HPF была разработана организацией The High Performance FORTRAN Forum, в состав которой вошли представители промышленности, науки, образования.

В языках упреждающих вычислений параллелизм реализован с помощью параллельного выполнения вычислений еще до того, как их результат потребуется для продолжения выполнения программы. Примерами таких языков являются MultiLisp — параллельная версия языка Lisp, Qlisp и Mul-T.

Языки программирования в рамках модели параллелизма данных: C*, APL, UC, HPF и т. д. Язык PARLOG является параллельной версией языка Prolog. Emerald — это объектный язык для распределенных приложений, похожий на Java. Erlang — параллельный язык для приложений реального времени. Maisie — параллельный язык, основанный на C. NESL — простой и переносимый язык параллельного программирования. Phantom — параллельный интерпретирующий язык. Scheme — один из диалектов Lisp. Cilk -алгоритмический многопоточный язык.

Кратко перечислим и среды разработки параллельных программ. aCe -среда разработки и выполнения программ в рамках модели параллелизма данных. ADAPTOR — система, основанная на языке HPF. Arjuna — объектно-ориентированная система программирования распределенных приложений. CODE — визуальная система параллельного программирования.

При разработке параллельных программ в рамках модели параллелизма задач чаще всего используются специализированные библиотеки и системы параллельного программирования PVM и MPI. PVM существует для самых разнообразных платформ, имеются реализации для языков Java и Python. Включает разнообразные средства создания параллельных программ.

MPI — это спецификация, первый вариант которой был разработан специальным комитетом MPIF (Message Passing Interface Forum) в 1994 году. Она описывает основные особенности и синтаксис прикладного программного интерфейса параллельных программ. Хотя MPI не обладает некоторыми замечательными особенностями PVM, эта спецификация основана на согласованных стандартах и все чаще применяется для создания параллельных программ. Используется с языками программирования C, C++ и FORTRAN. Имеется несколько свободно распространяемых реализаций MPI, в том числе и для разных платформ:

- MPICH разработана исследователями из Аргоннской Национальной Лаборатории (США) и университета штата Миссисипи и применяется на различных платформах

от кластеров рабочих станций до симметричных мультипроцессоров;
- LAM — разработка суперкомпьютерного центра штата Огайо, которая поддерживается группой исследователей университета Нотр-Дам и используется для кластерных систем;
- CHIMP создана в Центре Параллельных Вычислений Эдинбурга и предназначена для кластеров;
- NT-MPICH — версия MPICH для Windows NT и Windows 2000;
- WMPI — версия MPICH для кластеров под управлением ОС Windows;
- MacMPI предназначена для компьютеров Macintosh.

В настоящее время разработана спецификация MPI-2, созданы реализации MPI, ориентированные на разработку программ на языке C++, объектно-ориентированный вариант OOMPI и т. д.

Вопросы и задания для самостоятельной работы

1. Разработайте параллельный алгоритм вычисления величины:

$$C = \sum_{k=1}^{N} A_k B_k .$$

Здесь *A* и *B* — одномерные массивы.

2. Разработайте параллельный алгоритм перемножения матриц:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}.$$

Читатель, неискушенный в математике, может считать, что *A* и *B* — это просто двумерные массивы, протяженность которых по обоим измерениям одинакова.

3. Разработайте параллельный алгоритм быстрой сортировки Хоара.

4. Проведите исследование о возможности создания параллельного алгоритма решения известной задачи "о расстановке восьми ферзей".

5. Разработайте параллельный алгоритм решения одномерной конечно-разностной схемы с трехточечным шаблоном. Перечислите факторы, влияющие на эффективность алгоритма.

6. Напишите обзор и дайте сравнительный анализ языков параллельного программирования.

7. Дайте подробную характеристику каждому из четырех этапов разработки параллельного алгоритма.

8. Дайте сравнительную характеристику моделям параллелизма данных и параллелизма задач.

9. Перечислите типы коммуникаций.

10. Напишите обзор о стратегиях планирования вычислений.

ГЛАВА 3.

Введение в параллельное программирование с использованием MPI

В этой главе мы познакомимся с "Интерфейсом Передачи Сообщений" (Message Passing Interface или, сокращенно, MPI). Разберем общую организацию и структуру одной из его реализаций — MPICH. Обсудим особенности программирования с использованием MPICH, в том числе организацию обмена данными. Рассмотрим структуру MPI-программы. Заключительная часть главы посвящена трансляции и выполнению MPI-программ.

Что такое MPI?

Мы уже знаем, что традиционной является последовательная модель программирования. Программист, работая в рамках этой модели, считает, что у него имеется абстрактный компьютер, состоящий из одного центрального процессора и памяти (как оперативной, так и внешней). На таком компьютере и выполняется последовательная программа, представляющая собой единое целое.

В модели передачи сообщений, которая является разновидностью параллельной модели программирования, архитектура компьютера отличается от фон-неймановской. В этом случае считают, что компьютер состоит из нескольких процессоров, каждый из которых снабжен своей собственной памятью. Процессор и его собственная оперативная память, т. е. компьютер, включенный в коммуникационную сеть, часто

называют *хост-машиной* (от английского host) или, на программистском жаргоне, просто "хостом". Параллельная программа в модели передачи сообщений представляет собой набор обычных последовательных программ, которые отрабатываются одновременно. Обычно, каждая из этих последовательных программ выполняется на своем процессоре и имеет доступ к своей, локальной, памяти. Очевидно, в таком случае требуется механизм, обеспечивающий согласованную работу частей параллельной программы. Части параллельной программы в процессе их выполнения будем называть *подзадачами.*

Для обеспечения согласованной работы подзадачи должны обмениваться между собой информацией. Информация может быть разной. Это могут быть данные, обработку которых выполняет программа, а могут быть и управляющие сигналы. В модели передачи сообщений пересылка данных и управляющих сигналов происходят с помощью сообщений. Обратим внимание читателя на то, что обмен происходит не через общую или разделяемую память, а через другие коммуникационные среды, поэтому данная модель ориентирована на вычислительные системы с распределенной памятью. Пересылка сообщений — их отправка и прием реализуются программистом с помощью вызова соответствующих подпрограмм из библиотеки передачи сообщений. Схематически модель передачи сообщений представлена на рис. 3.1.

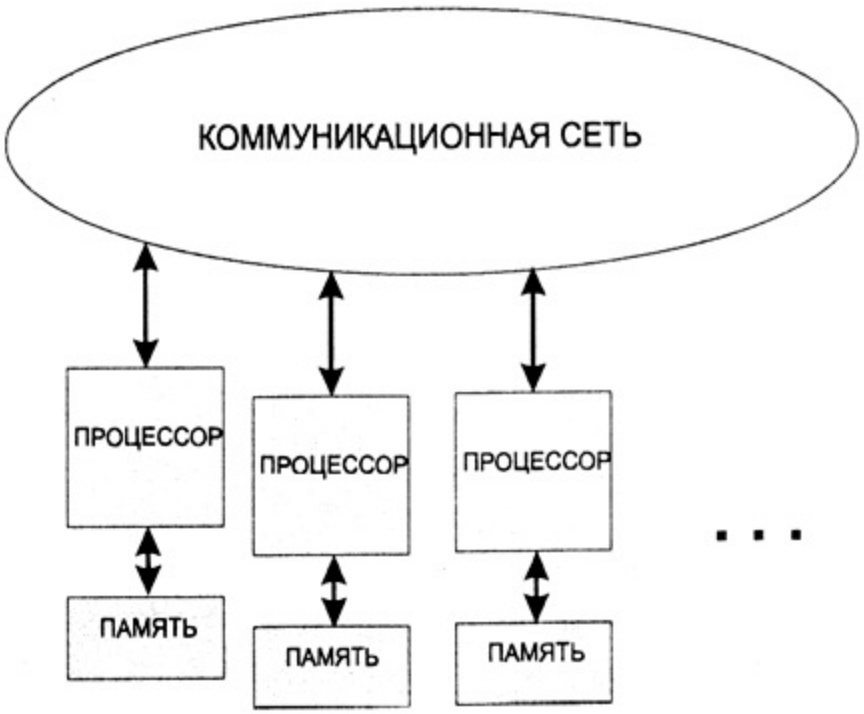


Рис. 3.1. *Модель передачи сообщений*

Модель передачи сообщений универсальна. Она может быть реализована на параллельных вычислительных, системах как с распределенной, так и с разделяемой памятью, на кластерах рабочих станций и даже на обычных, однопроцессорных компьютерах. В последнем случае, параллельная программа обычно выполняется в режиме отладки. Универсальность и независимость от архитектуры, скрытой от

программиста, стали одной из основных причин популярности модели передачи сообщений.

При разработке параллельного алгоритма, на этапе декомпозиции *(см. гл. 2),* исходная проблема разбивается на несколько частей-подзадач, каждая из которых часто может быть решена одним и тем же методом. Общий метод применяется к различным фрагментам набора данных, подлежащего обработке. В этом случае параллельная программа состоит из одинаковых фрагментов и на всех процессорах выполняются одинаковые подзадачи, фактически, одна и та же программа. Она может запускаться выделенной подзадачей, играющей роль "дирижера" (мы будем в дальнейшем называть ее *мастер-программой),* или другими подзадачами, входящими в параллельное приложение (см. листинг 3.1). Такая схема называется *SPMD-моделью* (Single Program Multiple Data) и является частным случаем модели передачи сообщений. В некоторых программных реализациях модели используется именно SPMD-вариант, а запуск нескольких подзадач на одном процессоре запрещен.

Листинг 3.1. Типичная структура программы в модели SPMD

program para

if (процесс = мастер) then

master else

slave endif end

В листинге 3.1 сначала каждый экземпляр программы уже в процессе своего выполнения определяет, является ли он мастер-программой. Затем, в зависимости от результата этой проверки, выполняется одна из ветвей условного оператора. Первая ветвь (master) соответствует мастер-задаче, а вторая (slave) — подчиненной задаче. Способы взаимодействия между подзадачами определяются программистом.

Сообщение *(см. гл. 1)* содержит пересылаемые данные и служебную информацию:

- идентификатор процесса-отправителя сообщения. В MPI идентификатор процесса называют его рангом;
- адрес, по которому размещаются пересылаемые данные процесса-отправителя;
- тип пересылаемых данных;
- количество данных (размер буфера сообщения — для того, чтобы принять сообщение, процесс должен отвести для него достаточный объем оперативной памяти!);
- идентификатор процесса, который должен получить сообщение;
- адрес, по которому должны быть размещены данные процессом-получателем.

Часть служебной информации составляет "конверт" сообщения. В "конверте" содержатся:

- ранг источника;
- ранг адресата;
- тег сообщения;
- идентификатор коммуникатора, описывающего область взаимодействия, внутри которой происходит обмен.

Эти данные позволяют адресату различать сообщения. Ранг источника дает возможность различать сообщения, приходящие от разных процессов. Тег -это задаваемое пользователем целое число от 0 до 32 767, которое играет роль идентификатора сообщения и позволяет различать сообщения, приходящие от одного процесса. Теги могут использоваться и для соблюдения определенного порядка приема сообщений. Данные, содержащиеся в сообщении, в общем случае организованы в массив элементов, каждый из которых имеет определенный тип.

Кроме пересылки данных система передачи сообщений должна поддерживать пересылку информации о состоянии процессов коммуникации. Это может быть, например, уведомление о том, что прием данных, отправленных другим процессом, завершен.

Перед использованием процедур передачи сообщений программа должна "подключиться" к системе обмена сообщениями. Подключение выполняется с помощью соответствующего вызова процедуры из библиотеки. В одних реализациях модели допускается только одно подключение, а в других -несколько подключений к системе.

Прием сообщения начинается с подготовки буфера достаточного размера. В этот буфер записываются принимаемые данные. Данные могут быть закодированы, в этем случае при их считывании выполняется декодирование. Кодирование данных, т. е. их преобразование в некоторый универсальный формат, требуется для того, чтобы обеспечить обмен сообщениями между системами с разным форматом представления данных.

Операция отправки или приема сообщения считается завершенной, если программа может вновь использовать такие ресурсы, как буферы сообщений.

Операции обмена сообщениями

Двухточечный (point-to-point) обмен — это простейшая форма обмена сообщениями. Называется он так потому, что в нем участвуют только два процесса, процесс-отправитель и процесс-получатель — источник и адресат.

Имеется несколько разновидностей двухточечного обмена:

- *синхронный обмен,* который сопровождается уведомлением об окончании приема сообщения;
- *асинхронный обмен,* который таким уведомлением не сопровождается;
- *блокирующие прием/передача,* которые приостанавливают выполнение процесса на время приема сообщения (рис. 3.2);
- *неблокирующие прием/передача,* при которых выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения.

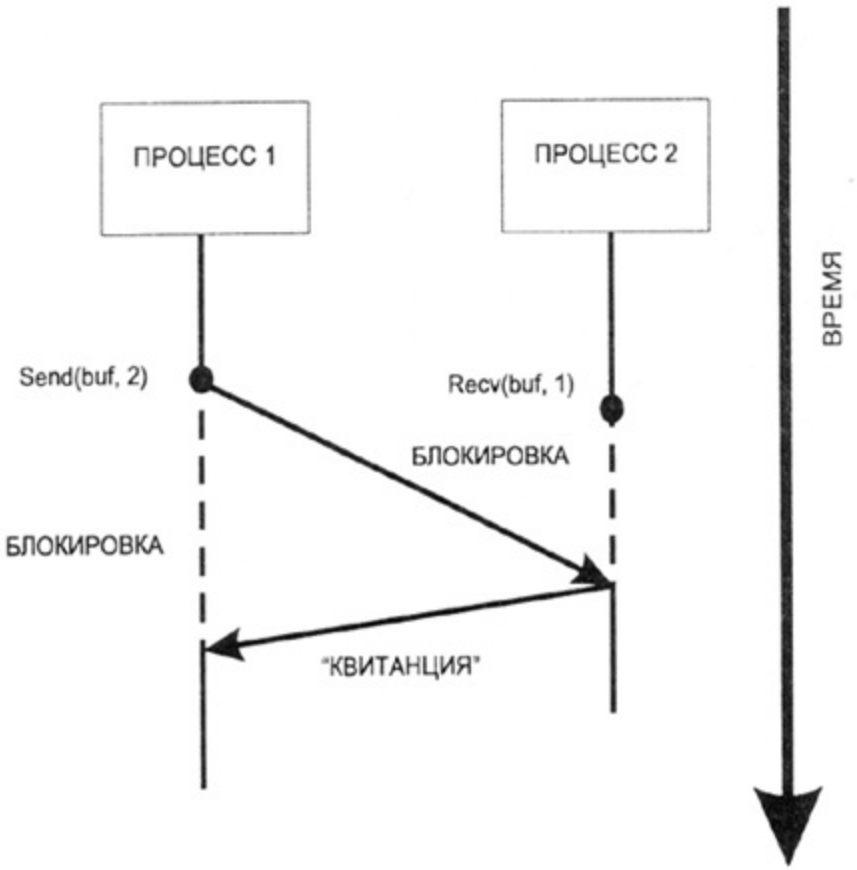


Рис. 3.2. *Блокирующий обмен сообщениями*

Неблокирующий обмен требует аккуратности при использовании функций приема. Поскольку неблокирующий прием завершается немедленно, для системы неважно, прибыло сообщение к месту назначения или нет. Убедиться в этом можно с помощью функций проверки получения сообщения. Обычно вызов таких функций размещается в цикле, который повторяется до тех пор, пока функция проверки не вернет значение "истина" (проверка получения прошла успешно). После этого можно вызывать функцию приема сообщения из буфера сообщений (рис. 3.3).

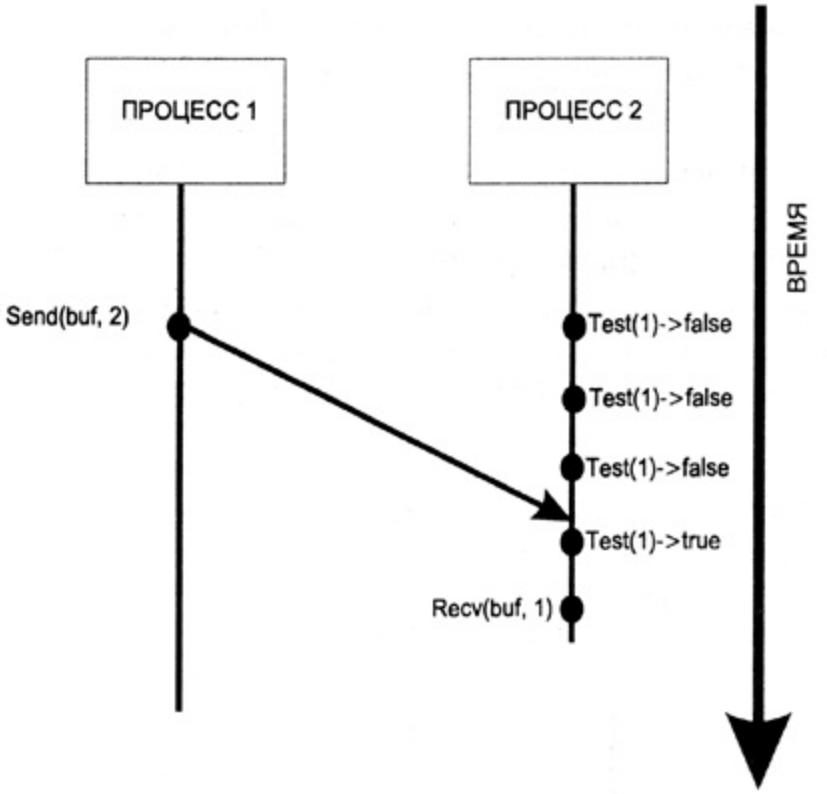


Рис. 3.3. *Неблокирующий обмен сообщениями*

В операции *коллективного* обмена вовлечены не два, а большее число процессов. Разновидностями коллективного обмена являются:

- *широковещательная передача* — выполняется от одного процесса ко всем;
- *обмен с барьером* — это форма синхронизации работы процессов, когда обмен сообщениями происходит только после того, как к соответствующей процедуре обратилось определенное число процессов;
- *операции приведения* — входными являются данные нескольких процессов, а результат — одно значение, которое становится доступным всем процессам, участвующим в обмене.

Важное свойство системы передачи сообщений состоит в том, гарантирует ли она сохранение порядка приема сообщений. Если гарантирует, то при отправке одним процессом другому нескольких сообщений они будут получены в той же последовательности, в которой были отправлены. Большинство реализаций модели передачи сообщений обладает данным свойством, но не во всех режимах обмена.

Имеются три способа реализации модели передачи сообщений:

- создание специализированного языка параллельного программирования;
- расширение обычного последовательного языка путем включения в него средств обмена сообщениями;
- использование специализированных библиотек в программах, написанных на

обычных языках последовательного программирования.

Язык Occam, разработанный для транспьютерных систем, является примером первого подхода. Пример второго подхода — языки CC+ (Compositional C++ — расширение C++) и FORTRAN M (расширение языка FORTRAN).

Имеется множество библиотек передачи сообщений, как свободно распространяемых, так и коммерческих, предназначенных для конкретных платформ. Чаще всего используются независимые от платформы (платформой называют сочетание архитектуры компьютера и установленной на нем операционной системы) библиотеки: PVM (Parallel Virtual Machine — "Параллельная Виртуальная Машина") и различные реализации MPI (Message Passing Interface). Пример платформозависимой библиотеки — библиотека для вычислительной системы nCUBE. Конкретные наборы функций в этих библиотеках могут различаться, но базовый набор примерно одинаков.

MPI — "Интерфейс Передачи Сообщений"

Практическое воплощение модель передачи сообщений нашла в спецификации, которая получила название *Интерфейс Передачи Сообщений*— MPI. Эта спецификация была разработана в 1993—1994 годах группой MPI Forum, в состав которой входили представители академических и промышленных кругов. Она стала первым стандартом систем передачи сообщений. В MPI были учтены достижения других проектов по созданию систем передачи сообщений: NX/2, Express, nCUBE, Vertex, p4, PARMACS, PVM, Chameleon, Zipcode, Chimp и т. д. Ее реализации представляют собой библиотеки подпрограмм, которые могут использоваться в программах на языках C/C++ и FORTRAN. В настоящее время принята новая версия спецификации - MPI-2.

В модели программирования, которую поддерживает MPI, программа порождает несколько процессов, взаимодействующих между собой с помощью обращений к подпрограммам передачи и приема сообщений. Обычно, при инициализации MPI-программы создается фиксированный набор процессов, причем каждый процесс выполняется на своем процессоре. В этих процессах могут выполняться разные программы, поэтому модель программирования MPI иногда называют *MPMD-моделью* (Multiple Program Multiple Data — множество программ множество данных), в отличие от SPMD-модели, где на каждом процессоре выполняются только одинаковые задачи.

Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций. При выполнении глобальных операций применяются коллективные обмены. Асинхронные коммуникации реализуются с помощью запросов о получении сообщений. Механизм, который называется *коммуникатором,* скрывает от программиста внутренние коммуникационные структуры.

Алгоритмы, в которых имеется фиксированное число подзадач, допускают прямую реализацию с помощью двухточечных и групповых обменов. Алгоритмы, в которых количество процессов изменяется в процессе выполнения программы, не могут быть реализованы в MPI непосредственно (в спецификации MPI-2 такая возможность уже есть). В этом случае приходится сразу, в момент запуска приложения, создавать множество процессов со структурой, в которую вписываются все возможные конфигурации подзадач, возникающие в процессе выполнения программы. Динамика будет в этом случае поддерживаться изменяющейся структурой коммуникаций. Часто это оказывается возможным.

Спецификация MPI обеспечивает переносимость программ на уровне исходных кодов и большую функциональность. Поддерживается работа на гетерогенных кластерах и симметричных многопроцессорных системах: Не поддерживается, как уже отмечалось, запуск процессов во время выполнения MPI-программы. В спецификации отсутствуют описания параллельного ввода/вывода и отладки параллельных программ. Эти возможности могут быть включены в состав конкретной реализации MPI в виде дополнительных пакетов и утилит. Совместимость разных реализаций MPI не гарантируется.

В *главе 2* мы уже говорили о таком важном свойстве параллельной программы, как детерминизм — программа должна всегда давать один и тот же результат для одного и того же набора входных данных. Модель параллельного программирования с передачей сообщений, вообще говоря, этим свойством не обладает, поскольку порядок получения сообщений от двух процессов третьим не определен. Если же один процесс последовательно посылает несколько сообщений другому процессу, MPI гарантирует, что второй процесс получит их именно в том порядке, в котором они были отправлены. Ответственность за обеспечение детерминированного выполнения программы ложится на программиста.

Организация MPICH

MPICH (MPI CHameleon) представляет собой одну из реализаций спецификации MPI, которая поддерживает работу на большом числе платформ и с различными коммуникационными интерфейсами, в том числе TCP/IP.

Основные особенности MPICH версии 1.2.2:

- полная совместимость со спецификацией MPI-1;
- наличие интерфейса в стиле MPI-2 с функциями для языка C++ из спецификации MPT-1;
- наличие интерфейса с процедурами языка FORTRAN-77/90;
- имеется реализация для Windows NT, которая распространяется в исходных текстах. Ее установка и использование отличаются от UNIX;
- поддержка большого числа архитектур, в том числе кластеров рабочих станций,

симметричных многопроцессорных систем и т. д.;
- частичная поддержка спецификации MPI-2;
- частичная поддержка параллельных ввода/вывода (ROMIO);
- наличие средств трассировки и протоколирования (на основе масштабируемого формата log-файлов SLOG);
- наличие средств визуализации производительности параллельных программ (upshot и jumpshot);
- наличие в составе MPICH тестов производительности и проверки функционирования системы.

В данной книге мы будем знакомиться с реализацией MPICH для наиболее распространенной операционной системы Linux. Популярность данной версии отчасти связана с тем, что она является некоммерческой и распространяется бесплатно.

Имеются и коммерческие варианты MPICH, которые выпускаются, как правило, с оптимизацией для конкретной платформы и могут содержать дополнительные функции и утилиты. Есть варианты для других операционных систем. В качестве примера можно привести WMPI -- реализацию MPICH для Microsoft Windows.

К числу недостатков MPICH можно отнести невозможность запуска процессов во время выполнения программы и отсутствие средств мониторинга (наблюдения) за текущим состоянием вычислительной системы. В результате этого, если происходит, например, аппаратный сбой на одном из процессоров, участвующих в выполнении параллельной MPI-программы, ее работа завершается аварийно. Невозможность динамического изменения набора процессов не позволяет использовать ресурсы вычислительной системы с максимальной эффективностью.

Если MPICH при установке был сконфигурирован для работы на кластере, то даже при пересылках сообщений на одном компьютере будет использоваться утилита rsh и сетевой протокол TCP/IP. Это не самое эффективное решение. Если же сборка производилась для системы с разделяемой памятью, MPICH-программа не сможет работать на кластере.

В состав MPICH входят библиотечные и заголовочные файлы. MPICH содержит более сотни подпрограмм. С пакетом MPICH поставляются средства визуальной отладки и профилирования параллельных программ. Это jump-shot или более старая версия upshot. Написаны они на языке Java и работают с файлами-протоколами событий (tracefiles) CLOG (jumpshot 2) и SLOG (jumpshot 3). Размер файла-протокола в третьей версии jumpshot может быть очень большим, до гигабайта. Имеется документация к MPICH в форматах HTML и PostScript.

Файлы MPICH в операционной системе Linux размещаются обычно в одном из подкаталогов системного каталога /usr, например в каталоге /usr/local/mpich. Этот каталог определяется системным администратором в момент установки пакета.

Заметим, что обычный пользователь также может установить MPICH, но только в своем домашнем каталоге. Кратко опишем структуру каталога MPICH и его содержимое:

- /usr/local/mpich/ — содержит файлы и подкаталоги MPICH;
- /usr/local/mpich/COPYRIGHT - файл с описанием авторских прав на пакет;
- /usr/local/mpich/README — заметки и инструкции по использованию пакета;
- /usr/local/mpich/bin/ — исполняемые файлы;
- /usr/local/mpich/examples/ — примеры программ;
- /usr/local/mpich/doc/ — документация по установке и работе с MPICH;
- /usr/local/mpich/inciude/ - заголовочные файлы (в том числе mpi.h и mpif.h);
- /usr/local/mpich/lib/ — библиотечные файлы;
- /usr/local/mpich/src/ — исходные тексты системы;
- /usr/local/mpich/man/ — справочные страницы по подпрограммам и утилитам MPICH.

Имеются и другие подкаталоги. В состав MPICH включены также примеры программ, которые располагаются в каталогах:

- mpich/examples/basic/ — демонстрация основных возможностей MPICH;
- mpich/examples/test/ — тестовые программы;
- mpich/examples/perftest/ — тестовые программы для определения производительности.

Набор примеров может быть разным в различных версиях MPICH.

Среди множества подпрограмм MPI можно выделить всего 6 основных, тех, которые используются чаще всего:

1. MPI_INIT(int *argc, char **argv) — подключение к **MPI.** Аргументы argc и argv требуются только в программах на С, где они задают количество аргументов командной строки запуска программы и вектор этих аргументов. Данный вызов предшествует всем прочим вызовам подпрограмм MPI.

2. MPI_FINALIZE ( ) — завершение работы с MPI. После вызова данной подпрограммы нельзя вызывать подпрограммы MPI. MPI_FINALISE должны вызывать все процессы перед завершением своей работы.

3. MPi_COMM__size(comm, size) — определение размера области взаимодействия. Здесь соотп — входной параметр-коммуникатор, а выходным является параметр size целого типа, количество процессов в области взаимодействия.

4. MPi__CoMM_RANK(comm, pid) — определение номера процесса. Здесь pid - идентификатор процесса в указанной области взаимодействия.

5. MPI_SEND(buf, *count,* datatype, dest, tag, comm) **— отправка сообщения.**

Все параметры являются входными: buf — адрес буфера отправки, count — количество пересылаемых элементов данных (неотрицательное целое значение), datatype — тип пересылаемых данных, tag — тег сообщения (целое значение), comm — коммуникатор.

6. MPI_RECV(buf, count, datatype, source, tag, comm, status) — Прием сообщения. Параметр buf выходной, это адрес получения буфера, status тоже выходной параметр — статус завершения операции, source — идентификатор процесса, от которого получаем сообщение (входной параметр). Все остальные параметры — входные и их назначение совпадает с назначением аналогичных параметров MPI_SEND.

Здесь приведено условное описание интерфейса к этим подпрограммам. Спецификатор source в функции MPI_RECV позволяет программисту указать, что сообщение должно быть получено либо только от конкретного процесса, задаваемого его целочисленным идентификатором, либо от любого процесса. В последнем случае используется специальное значение MPI_ANY_ SOURCE. Первый способ предпочтительнее, поскольку он исключает ошибки, связанные с неопределенностью порядка поступления данных.

С точки зрения коммуникаций подпрограммы можно разделить на две группы: *локальные* и *нелокальные.* Локальные подпрограммы не выполняют пересылок данных. Они используют данные текущего процесса или ожидают поступления сообщения.

Область взаимодействия (область связи) определяет группу процессов. Все процессы, принадлежащие одной области взаимодействия, могут обмениваться сообщениями. Описывает это множество процессов специальная информационная структура, которая называется *коммуникатором.*

Коммуникатор описывает контекст коммуникации для операции обмена. Каждый контекст задает отдельную область взаимодействия, сообщения принимаются в том контексте, в котором они были отправлены, а сообщения, отправленные в разных контекстах, не пересекаются и не мешают друг другу. Другими словами, процессы, связанные с MPI-программой, могут взаимодействовать, только если они связаны с одним коммуникатором (рис. 3.4). Значение коммуникатора (оно используется по умолчанию) MPI_COMM_WORLD соответствует всем процессам данной программы. Всем процессам в области взаимодействия присваиваются целые положительные номера от 0 до некоторого максимального, а номер текущего процесса можно определить с помощью вызова MPI_COMM_RANK.

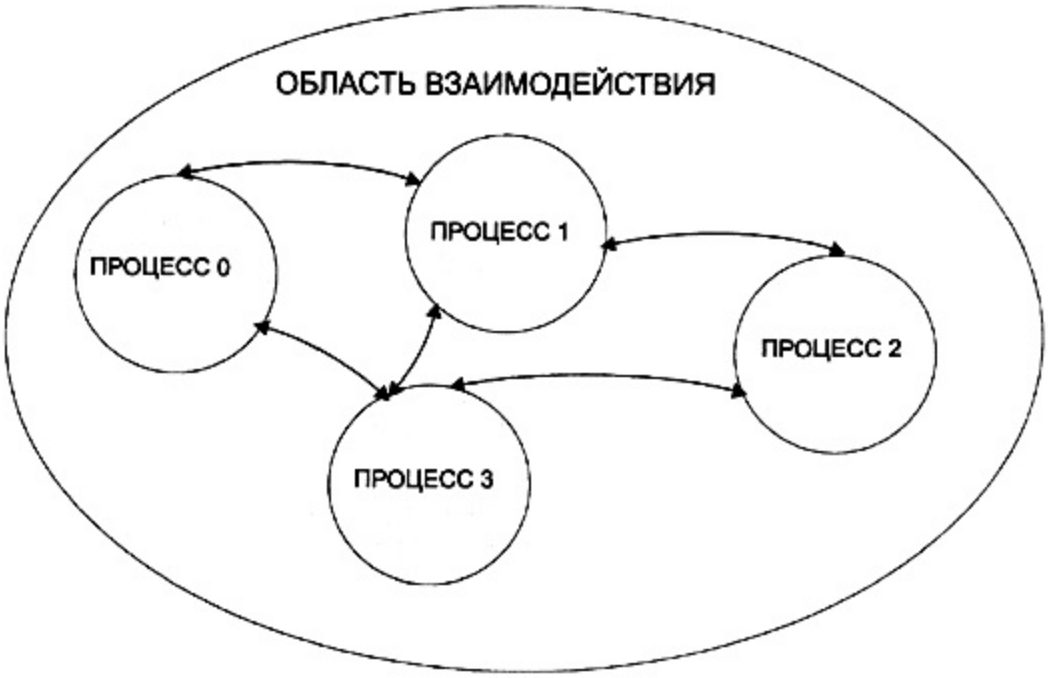


Рис. 3.4. *Область взаимодействия MPI-программы*

Из процессов, входящих в существующую область взаимодействия, могут создаваться новые области взаимодействия. Нумерация процессов в разных областях взаимодействия независима. Коммуникатор описывает область взаимодействия. Для одной области могут существовать несколько коммуникаторов. Стандартный коммуникатор MPI_COMM_WORLD создается автоматически. Далее в книге мы будем для краткости называть коммуникатором область взаимодействия процессов, однако читатель должен помнить, что это разные вещи.

Коммуникаторы являются константами типа MPI_Comm в программах на языке C и типа INTEGER в языке FORTRAN. Кроме упоминавшегося MPI_COMM_WORLD имеются также: MPI_COMM_SELF — коммуникатор, содержащий только вызывающий процесс, и MPI_COMM_NULL — пустой коммуникатор.

Номер процесса называется его *рангом.* Ранг используется для указания конкретного процесса, например, при пересылке сообщений. Вообще говоря, процесс может принадлежать нескольким областям взаимодействия и в каждой у него будет свой собственный ранг.

Пример простейшей MPI-программы на некоем условном языке приведен в листинге 3.2.

Листинг 3.2. Пример простейшей MPI-программы

```
program mpi_example

MPI_INIT()
```

```
MPI_COMM_SIZE(MPI_COMM_WORLD, count)

MPI_COMM_RANK(MPI_COMM_WORLD, id)

write(id, count)

MPI_FINALIZE()

end
```

В этой программе сначала происходит подключение к MPI, затем определяется количество процессов, входящих в область взаимодействия, номер процесса, соответствующего данному экземпляру программы, полученные значения выводятся и программа завершает свою работу, предварительно отключившись от MPI.

Привязка к языку С

Библиотека MPICH может использоваться в программах на языках FORTRAN и C/C++. Вызов подпрограмм в этих двух случаях различается. По-разному указываются имена функций, различаются способы получения кодов завершения подпрограмм и т. д.

При использовании MPI в программах на языке С в именах функций используется префикс MPI_, а первая буква имени набирается в верхнем регистре. Не рекомендуется использовать имена, которые начинаются с MPI_ для своих функций или переменных. Согласно спецификации имена подпрограмм Имеют вид Класс_действие_подмножество или класс_действие. Аналогичное правило действует и для подпрограмм MPI для языка FORTRAN. В C++ подпрограмма является методом для определенного класса, имя имеет в этом случае вид MPI: :Класс: :действие_подмножество. Для некоторых действий введены стандартные наименования: Create — создание нового объекта, Get — получение информации об объекте, set — установка параметров объекта, Delete — удаление информации, is — запрос о том, имеет ли объект указанное свойство.

Значения кода завершения имеют целый тип и определяются по значению функции. Имена констант MPI записываются в верхнем регистре. Их описания находятся в заголовочном файле mpi.h, который обязательно включается в MPI-программу. Имя этого файла может быть иным в других реализациях MPI. Входные параметры функций передаются по значению, а выходные (и INOUT) — по ссылке. Переменная status имеет тип MPi_status и является структурой с полями status.MPI_SOURCE и status.MPI_TAG. В MPI принята своя система обозначения типов данных, которая соответствует ти-

пам данных в языках С (табл. 3.1) и FORTRAN (табл. 3.2), хотя соответствие это неполное.

Таблица 3.1. *Типы данных MPI для языка C*

Тип данных MPI	**Тип данных C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Нет соответствия
MPI_PACKED	Нет соответствия

В MPI должны соблюдаться правила совместимости типов. Соответствие типов должно, как правило, иметь место в процедурах отправки и процедурах приема сообщений. Из базовых типов могут быть сконструированы более сложные типы данных.

Привязка к языку FORTRAN

В языке FORTRAN имена подпрограмм набираются в верхнем регистре (регистр букв в именах не имеет значения, однако имеются определенные соглашения и традиции, соблюдение которых желательно). Все имена подпрограмм и констант MPI начинаются с MPI_. Коды завершения передаются через дополнительный параметр целого типа (он обычно находится на последнем месте в списке параметров подпрограммы). Код успешного завершения — MPI_SUCCESS. Константы и другие объекты MPI описываются в файле mpi.h, который обязательно включается в MPI-программу с помощью оператора include. Этот оператор находится в начале программы. Переменная status является массивом стандартного целого типа. Его размер (MPI_STATUS_SIZE) и индексы задаются именованными константами:

integer status(MPI_STATUS_SIZE) if (status(MPI TAG).EQ.tagl) then

Таблица 3.2. *Типы данных MPI для языка FORTRAN*

Тип данных MPI	**Тип данных FORTRAN**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MP I_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_COMPLEX	COMPLEX
MPI _LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER

MPI_BYTE	Нет соответствия
MPI_PACKED	Нет соответствия
MPI_INTEGER1	INTEGER*1
MPI _ INTEGER2	INTEGER* 2
MPI_INTEGER4	INTEGER* 4
MPI_REAL4	REAL* 4
MPI_ REALS	REAL* 8

В программах на языке FORTRAN такие объекты MPI, как MPi_Datatype или MPI_comm — целого типа (INTEGER).

Итак, в программах на языке С используются библиотечные функции MPI, а в программах на языке FORTRAN — процедуры. В дальнейшем и те и другие мы будем называть подпрограммами MPI.

Коды завершения

Коды завершения возвращаются в качестве значения функции С или через последний аргумент процедуры FORTRAN. Исключение составляют подпрограммы MPI_wtime и MPI_wtick, в которых возвращение кода ошибки не предусмотрено.

Используются стандартные значения MPI_SUCCESS — при успешном завершении вызова и MPI_ERR_OTHER — обычно при попытке повторного вызова процедуры MPI_Init.

Системой распознаются и другие ошибки. Вместо числовых кодов в программах обычно используют специальные именованные константы. Среди них:

- MPI_ERR_BUFFER — неправильный указатель на буфер;
- MPI_ERR_COMM — неправильный коммуникатор;
- MPI_ERR_RANK — неправильный ранг;

- MPI_ERR_OP — неправильная операция;
- MPI__ERR_ARG — неправильный аргумент;
- MPI_ERR_UNKNOWN — неизвестная ошибка;
- MPI_ERR_TRUNCATE — сообщение обрезано при приеме;
- MPI_ERR_INTERN — внутренняя ошибка. Обычно возникает, если системе не хватает памяти.

Как устроена MPI-программа

В программе MPI следует соблюдать определенные правила, без которых она окажется неработоспособной. Прежде всего, в начале программы, сразу после ее заголовка, необходимо подключить соответствующий заголовочный файл. В профамме на языке C это mpi.h:

#include "mpi.h"

а в профамме на языке FORTRAN — mpif.h:

include "mpif.h"

В этих файлах содержатся описания констант и переменных библиотеки MPI.

Первым вызовом библиотечной процедуры MPI в профамме должен быть вызов подпрофаммы инициализации MPi_init, перед ним может располагаться только вызов MPI_Initialized, с помощью когда определяют, инициализирована ли система MPI. Вызов процедуры инициализации выполняется только один раз. В языке FORTRAN у процедуры инициализации единственный аргумент — код ошибки:

integer IERR

CALL MPI_INIT(IERR)

В языке C параметры функции инициализации получают адреса аргументов главной программы, задаваемых при ее запуске:

MPI_Init(&argc, &argv);

Загрузчик mpirun в конец командной строки запуска MPI-программы добавляет служебные параметры, необходимые MPi_init. В программах на языке FORTRAN аргументы командной строки не используются.

Процедура инициализации создает коммуникатор со стандартным именем MPI_COMM_WORLD. Это имя указывается во всех последующих вызовах процедур MPI.

После выполнения всех обменов сообщениями в программе должен располагаться вызов процедуры MPi_Finaiize(ierr). В результате этого вызова удаляются структуры данных MPI и выполняются другие необходимые действия. Программист должен позаботиться о том, чтобы к моменту вызова процедуры MPi_Finaiize были завершены все пересылки данных. После выполнения данного вызова другие вызовы процедур **MPI,** включая MPI_init, недопустимы. Исключение составляет подпрограмма MPi_Initiaiized, которая возвращает значение "истина", если процесс вызывал MPi_Init. Данный вызов может находиться в любом месте программы.

В тех частях программы, которые находятся до вызова MPi_iinit и после вызова MPi_Finaiize, не рекомендуется открывать файлы, выполнять считывание и запись в файлы стандартного ввода и вывода.

Выполнение всех процессов в коммуникаторе может быть прервано процедурой MPi_Abort. Если коммуникатор описывает область связи всех процессов программы, будет завершено выполнение программы. Данный вызов используется для аварийного останова программы при возникновении серьезных ошибок. Примеры простейших программ на языках FORTRAN и C приведены в листингах 3.3 и 3.4.

Листинг 3.3. Простейшая MPI-программа на языке FORTRAN

```
PROGRAM MAINJMPI

INCLUDE 'mpif.h'

INTEGER MYID, NUMPROCS, IERR

CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NUMPROCS, IERR)

PRINT *, "Process ", myid, " of ", numprocs

CALL MPI_FINALIZE(IERR)

STOP

END
```

Листинг 3.4. Простейшая MPI-программа на языке C

```
#include "mpi.h"

#include <stdio.h>
```

```
int main(int argc,char *argv[])

{

int myid, numprocs;

MPI_Init (Sargc, sargv) ;

MPI_Comm_size (MPI_COMM_WORLD, unumprocs) ;

MPI_Comm_rank (MPI_COiyiM_WORLD, Smyid) ;

fprintf (stdout, "Process %d of %d\n", myid, numprocs)

MPI_Finalize() ;

return 0;

}
```

В Обеих программах используются подпрограммы

MPI_Comm_size И MPI_Comm_

rank. Напомним, что первая из них возвращает количество процессов, входящих в область взаимодействия, коммуникатор которой указан в качестве ее первого аргумента. Вторая подпрограмма определяет ранг процесса.

Программа написана, что дальше?

Трансляция

Утилиты трансляции и сборки находятся в подкаталоге /usr/local/mpich/bin. Этот каталог рекомендуется включить в путь поиска исполняемых файлов. В различных операционных системах это делается по-разному. Утилиты трансляции запускают трансляторы обычных языков C и FORTRAN, а в командную строку их запуска подставляют ссылки на необходимые библиотечные и заголовочные файлы. Подключаться могут также библиотеки, используемые для отладки и профилирования.

Для трансляции и компоновки программ на языке C++ используется команда mpiCC, а для трансляции программ на языке C — команда mpicc (напомним, что при работе в операционной системе UNIX регистр букв в команде важен). Командные файлы трансляции и компоновки можно и не использовать, однако при этом придется явно указывать многочисленные ключи, командная строка при этом будет громоздкой.

Перечень основных ключей командной строки:

- -mpilog — сборка версии программы, которая создает протоколы для утилит профилирования;
- -mpitrace — сборка версии программы, которая генерирует трассировочные файлы;
- -mpianim — сборка версии программы, которая генерирует анимирован-ное изображение процесса выполнения программы;
- -show — показать команды трансляции и сборки, не выполняя их;
- -help — вывод краткой информации о команде;
- -echo — вывод на экран команд по мере их выполнения.

Информацию об остальных ключах можно найти на справочных страницах MPICH (для их просмотра применяют команду man). Пример использования команды:

mpicc -o fft f f t.c

Для трансляции и компоновки программ на языке FORTRAN-77 и FORTRAN-90 применяются команды mpif77 и mpif90. Ключи, используемые с этими командами, те же, что и для команд mpicc и mpicc.

Пользователь может назначить свой собственный транслятор, определив значения переменных окружения MPICH_CC, MPicH_F77, MPICH_CCC или MPiCH_F90. При этом следует помнить, что трансляторы должны быть совместимы с теми, которые используются по умолчанию. Переназначить программу-компоновщик можно с помощью переменных окружения MPICH_

CLINKER, MPICH_F77LINKER, MPICH_CCLINKER И MPICH_F90LINKER.

Транслятор, библиотеки и ключи, необходимые для нормальной сборки MPI-программы, можно узнать с помощью ключей -compile_info и -link_info. Примеры выполнения команд mpicc и mpif77 с указанными ключами приводятся ниже:

mpicc -compile_info

cc -DUSE_STDARG -DHAVE_STDLIB_

H=1 -DHAVE_STRING_

H=1 -DHAVE_UNISTD_H=1

-DHAVE_STDARG__

H=1 -DUSE_STDARG=1 -DMALLOC_

RET_VOID=1 -I/usr/local/mpich/include -c

```
# mpicc -link_info

CC -DUSE_STDARG -DHAVE_STDLIB_

H=1 -DHAVE_STRING_

H=1 -DHAVE_UNISTD_

H=1 -DHAVE_STDARG_

H=1 -DUSE_STDARG=1 -DMALLOC_

RET_VOID=1 -L/usr/local/mpich/lib -lmpich

# mpif77 -compile_info

f77 -I/usr/local/mpich/include -c

# mpif77 -link_info

f77 -L/usr/local/mpich/lib -lmpich
```

"Устройства" MP1CH

В MPICH используется "интерфейс абстрактных устройств" (ADI — Abstract Device Interface), который обеспечивает переносимость системы на различные платформы. "Устройства" представляют собой наборы процедур, реализующих пересылку и прием пакетов с учетом конкретной архитектуры. Устройство ch_p4, например, основано на библиотеке p4, которая поддерживает разработку параллельных программ для различных платформ. Для модели программирования с разделяемой памятью p4 дает программисту набор примитивов, из которых создаются более сложные программные конструкции. Для модели с распределенной памятью p4 дает реализации процедур пересылки данных и некоторые другие операции. Обе модели можно объединить в одну "кластерную" модель программирования. Библиотека p4 поддерживает большое количество архитектур. Список поддерживаемых архитектур может быть без большого труда дополнен.

Выполнение параллельной программы

Запуск MPI-программы на выполнение не стандартизован, поэтому в различных реализациях интерфейса могут использоваться разные способы. Для выполнения MPI-программ в MPICH используется загрузчик приложений mpirun. Это командный файл, который выполняет необходимые подготовительные действия, такие как определение конфигурации вычислительной системы, и запускает указанное количество копий программы. Команда выглядит следующим образом:

mpirun -np n [ключи MPI] программа [ключи и аргументы программы]

где л — число запускаемых процессов, которые обычно являются копиями одной и той же программы.

Список ключей MPI (ключи указываются перед именем исполняемого файла программы) приведен в табл. 3.3.

Таблица 3.3. *Основные ключи загрузчика приложений mpirun*

Ключ	**Описание**
-arch <архитектура>	Архитектура параллельной вычислительной системы, которая должна соответствовать суффиксу в имени файла тасЫпез. <архитектура>
-h -machine <имя компьютера>	Краткая информация о команде Использовать процедуру запуска, специфическую для указанного компьютера
-machinefile файл	Использовать список компьютеров из указанного файла
-nolocal	Не запускать программу на локальной машине (этот ключ работает только для устройства ch_p4)
-stdin файл	Использовать в качестве файла стандартного ввода программы указанный файл. Данный ключ применяется для задач, запускаемых в пакетном режиме (с системой очередей NQS)

-t	Режим тестирования. Программа не запускается, выводятся только действия, которые должны быть выполнены
-v	Вывод подробных сообщений о выполняемых шагах
-dbx	Запустить первый процесс с отладчиком dbx
-gdb	Запустить первый процесс с отладчиком gdb
-xxgdb	Запустить первый процесс с отладчиком xxgdb
-tv	Запустить программу с отладчиком totalview

Особенности выполнения программ на кластерах рабочих станций

На кластерах рабочих станций каждый процесс из параллельной программы должен запускаться индивидуально. Для этого требуется дополнительная информация, которая содержится в специальном файле. Если не используется устройство chameleon, перед запуском программы должен быть создан текстовый файл mpich/util/machmes/machines.<архНТеКТура> с перечнем сетевых имен компьютеров, на которых может запускаться программа. Здесь вместо <архитектура> подставляется название архитектуры. При использовании другого файла, его имя должно быть указано после ключа -machine команды mpirun. MPICH поддерживает работу со следующими устройствами:

- chameleon — универсальное устройство, включая chameleon/pvm, chame-leon/p4 и т. д.;
- p4 — устройство ch_p4 для кластера рабочих станций;
- ibmspx — устройство ch_eui для вычислительной системы IBM SP2;
- anlspx — устройство ch_eui для вычислительной системы ANL SPx;

- sgi_mp — устройство ch_shmem для многопроцессорных систем с разделяемой памятью, прежде всего SGI;
- smp — устройство ch_shmem для симметричных многопроцессорных систем;
- execer — применение произвольного командного файла для запуска программ с использованием ch_p4, но без файла procgroup;

и некоторыми другими.

После ключа -arch может быть указана архитектура вычислительной системы. Поддерживается работа со следующими платформами (в названиях важен регистр букв):

- sun4 - ОС SUN OS *4.x;*
- Solaris — ОС Solaris, в том числе solaris86 (Solaris для платформ Intel);
- hpux - ОС HP UX; O sppux - ОС SPP UX;
- rs6000 — ОС AIX для компьютера IBM RS6000; a sgi (IRIX) - ОС IRIX *4.x, 5.x* и *6.x;*
- sgi5 — ОС IRIX *5.x* для компьютеров R4400;
- alpha — DEC Alpha;
- intelnx — Intel i860 или Intel Delta;
- CRAY - CRAY XMP, YMP, C90, J90, T90; a cray_t3d - CRAY T3D;
- freebsd — персональные компьютеры под управлением ОС FreeBSD;
- netbsd — персональные компьютеры под управлением ОС NetBSD;
- LINUX — персональные компьютеры под управлением ОС Linux;
- LINUX_ALPHA — Alpha-компьютеры под управлением ОС Linux; и некоторыми другими.

Для многопроцессорных компьютеров Silicon Graphics под управлением ОС IRIX ключ -comm=ch_p4 запрещает использование коммуникаций через разделяемую память, а -comm=shared разрешает.

С помощью командного файла tstmachines из каталога /usr/local/mpich/sbin можно проверить, работают ли с MPICH все компьютеры, указанные в списке машин. Этот файл выполняет команду вывода содержимого каталога на удаленных машинах с помощью rsh (remote shell), что и является проверкой. Единственный аргумент данной команды - • название архитектуры. Если все работает нормально, вывод у команды отсутствует, если же нет, выводится соответствующее сообщение.

В командный файл tstmachines включены следующие проверки. Во-первых, может ли быть процесс запущен на удаленной машине и если нет, то по какой причине? Возможно, не установлен пакет r-utils или в домашнем каталоге пользователя отсутствует файл .rhosts. Следует учесть, что для последнего должны быть установлены права доступа только по записи и чтению и только владельцу файла.

Устройству ch_p4 для работы не требуется команда rsh, оно может использовать альтернативные методы коммуникации. Во-вторых, следует проверить, доступен ли текущий рабочий каталог на всех машинах? И, наконец, может ли пользовательская программа выполняться на удаленной системе? Для этого везде должны быть установлены необходимые динамические библиотеки, другие компоненты системы и все нужные части прикладной программы.

Ключи загрузчика mpirun для кластеров и ключи для запуска программ в пакетном режиме приведены в табл. 3.4.

Таблица 3.4. *Ключи загрузчика mpirun для запуска на кластере и в пакетном режиме*

Ключ	**Описание**
	Использовать для запуска программы на кластере execer
-e -рд	Использовать для запуска p4 программ файл procgroup вместо execer (по умолчанию)
-leave_pg	Не удалять после выполнения файл p4 procgroup
-p4рд файл	Использовать существующий файл p4 procgroup, не создавая новый
-tcppg *файл*	Использовать существующий файл tcp procgroup
-p4ssport *порт*	Использовать безопасный сервер p4 на указанном порте для запуска программ. Если номер порта нулевой, используется значение переменной окружения MPI P4SSPORT. Применение сервера может ускорить запуск процесса
	Ключи для пакетного режима
-mvhome	Перенести исполняемый файл в домашний каталог. Используется с anlspx
-mvback	

файлы	Перенести указанные файлы обратно в текущий каталог. Используется только с -mvhome, в противном случае не выполняет никаких действий
-maxtime *число*	Максимальное процессорное время выполнения в минутах. В настоящее время используется только с anlspx. Значение по умолчанию равно 15 минутам

Приложение, написанное в MPICH, может запускаться на выполнение с ключами командной строки. Эти ключи приведены в табл. 3.5.

Таблица 3.5. *Ключи командной строки запуска MP1-программы*

Ключ	**Описание**
-mpiqueue	Вывести состояние очередей сообщений при вызове MPI FINALIZE. Информация, поступающая от разных процессоров, может смешиваться, это не очень удобно
-mpiversion	Вывести версию реализации
-mpinice *число*	Увеличить значение относительного приоритета (nice number) на указанное значение
-mpedbg	Запустить в окне эмулятора X-терминала xterm отладчик, при возникновении ошибки. Данный ключ работает, только если MPICH был сконфигурирован с ключом -mpedbg
-mpimem	Если MPICH был скомпилирован с ключом -DMPIR DEBUG MEM, все операции размещения и освобождения памяти, внутренние для MPICH, будут проверяться

	на наличие ошибок, связанных с нарушением выделяемой памяти
-mpidb *ключи*	Активирует ключи отладки MPI (но не пользовательских программ): • mem— включить трассировку динамической памяти для внутренних объектов MPI; • memall — генерировать вывод всех операций размещения и освобождения памяти; • ptr— включить трассировку преобразований внутренних указателей MPI; • ref — трассировка использования внутренних объектов MPI; • ref file *файл*— записывать трассировку внутренних объектов MPI в файл; • trace — трассировка вызовов подпрограмм

Особенности выполнения программ MPICH на SMP-кластерах

На кластере, состоящем из симметричных многопроцессорных машин, пользователь может задавать максимальное количество процессов, запускаемых на каждом узле. Для этого устройство ch_p4 должно быть сконфигурировано с ключом -comm=shared. Количество процессов .указывается в файле со списком машин (machines) для каждого компьютера. Это число не должно превышать число процессоров на каждом узле. Формат файла простой — в каждой строке записывается сетевое имя компьютера, за которым следует разделитель ":" (двоеточие) и количество процессов (листинг 3.5).

Листинг 3.5. Пример файла machines

f01.ptc.spb.ru f02.ptc.spb.ru f03.ptc.spb.ru f04.ptc.spb.ru fserver.ptc.spb.ru:2

В примере из листинга 3.5 в список внесены четыре однопроцессорных машины (f01-f02) и одна двухпроцессорная (fserver). Изменить максимальное число процессов на каждом узле можно с помощью переменной окружения

MPI_MAX_CLUSTER_SIZE.

Особенности выполнения программ MPI на гетерогенных системах

Гетерогенные сети рабочих станций состоят из компьютеров с разной архитектурой, работающих под управлением различных операционных систем. В этом случае при запуске программы архитектуры задаются с помощью ключа -arch:

mpirun -arch LINUX -np 2 -arch LINUX_ALPHA -np 2 program.%a

Специальное имя программы program. %a позволяет определить разные имена исполняемых файлов, поскольку в данной ситуации программа, оттранслированная на компьютере под управлением ОС LINUX, не подходит для выполнения, скажем, на компьютерах SGI. Суффикс %a заменяется названием архитектуры, например, program. sun4. Файлы могут также располагаться в разных каталогах, например:

mpirun -arch sun4 -np 3 -arch IRIX -np 2 /tmpAa/program

Архитектура указывается перед ключом -np.

Сначала в командной строке задается главная программа.

При каждом запуске mpirun создает временный файл со списком машин. Для него используются данные из файла машин. Временному файлу автоматически присваивается имя PInnnn, где nnnn является идентификатором процесса. Если при запуске mpirun указать ключ -keep_pg, можно будет определить, где mpirun выполнял последние задачи.

Этот файл можно создать заранее и указать его в качестве аргумента mpirun. Для устройства ch_p4:

mpirun -p4pg pgfile myprog

где pgfile — имя файла.

Такой файл может быть полезен, если необходимо запустить разные исполняемые файлы на разных компьютерах или нужно запустить программу в кластере мультипроцессоров с разделяемой памятью и требуется указать число процессов на каждой машине. Он является текстовым и состоит из строк вида:

<сетевое имя компьютера>

<количество процессов>

<имя исполняемого файла> [<login>]

Пример:

f01 0 /home/sergei/myprog

f02 1 /home/andreevitch/myprog

f03 1 /home/nemnugin/myprog

Здесь определены три процесса, по одному на каждую рабочую станцию, причем имена домашних каталогов пользователя могут различаться. Нулевое значение в первой строке означает, что на данном процессоре пользователем с помощью командной строки может быть запущен только один процесс.

При работе с устройствами ch__p4, ch_mpl и globus2 можно запускать разные исполняемые файлы. Такой стиль программирования, как мы уже знаем, называют MPMD.

Отладка

С помощью программ clogxxx можно собрать данные о работе программы и просмотреть их с помощью утилит визуализации jumpshot. Запуск программы для просмотра протоколов событий выполняется командным файлом logviewer, который располагается в подкаталоге /usr/local/mpich/bin. Для различных реализаций MPI имеются средства отладки, созданные независимыми разработчиками, их обзор будет дан в *приложении 1.*

Вопросы и задания для самостоятельной работы

1. Сформулируйте характерные особенности модели передачи сообщений.

2. Дайте характеристику пакету MPICH. Опишите его возможности и ограничения.

3. Убедитесь, что на вашем компьютере установлен пакет MPICH, если нет, установите.

4. Познакомьтесь с "внутренним устройством" файлов mpicc, mpif90 и mpirun (для этого потребуется знание командного языка интерпретаторов bash и csh).

5. Наберите, откомпилируйте простейшую программу MPI (см. листинги 3.2 и 3.3) и попробуйте ее запустить на разном числе процессоров. Сделайте это для программ на языках C и FORTRAN.

6. Наберите, откомпилируйте и запустите программу, которая выводит на экран приветствие "Hi, Parallel Programmer!", если ранг процесса равен нулю.

ГЛАВА 4.

Обмен данными в MPI

В *главе 3* мы познакомились с тем, как устроена простейшая программа, написанная с использованием библиотеки MPICH. Наша первая MPI-программа, строго говоря, еще не является параллельной. Читатель уже знает, что "настоящая" параллельная программа во время выполнения представляет собой множество процессов (по крайней мере, два), в каждом из которых решается своя подзадача. Эти подзадачи связаны между собой обменом данными - промежуточными и конечными результатами работы. Наша MPI-программа пока умеет только "произносить" приветствие (выполняет вывод на терминал), может узнать свое "имя" (ранг процесса). Пришло время познакомиться со средствами обмена данными, которые имеются в MPICH, и написать первые действительно параллельные программы.

В этой главе мы познакомимся с основными типами обмена — *двухточечным* и *коллективным.* Рассмотрим особенности различных режимов обмена, разберем правила использования основных подпрограмм передачи и приема сообщений. Во второй части главы мы научимся создавать различные "сообщества" процессов —

группы, которые позволяют выделить часть области взаимодействия для выполнения обменов, ограниченных этой группой. В заключение приводятся задания для самостоятельной работы. Мы настоятельно рекомендуем читателю выполнять эти задания, поскольку самым лучшим "учителем" параллельного программирования является практика, которая позволяет привыкнуть к новой и, возможно, непривычной модели программирования.

Двухточечный обмен сообщениями

Двухточечный обмен, с точки зрения программиста, выполняется следующим образом (рис. 4.1): для пересылки сообщения процесс-источник вызывает подпрограмму передачи, при обращении к которой указывается ранг процесса-получателя (адресата) в соответствующей области взаимодействия. Последняя задается своим коммуникатором, обычно это MPI_COMM_WORLD. Процесс-получатель, для того чтобы получить направленное ему сообщение, должен вызвать подпрограмму приема, указав при этом ранг источника.

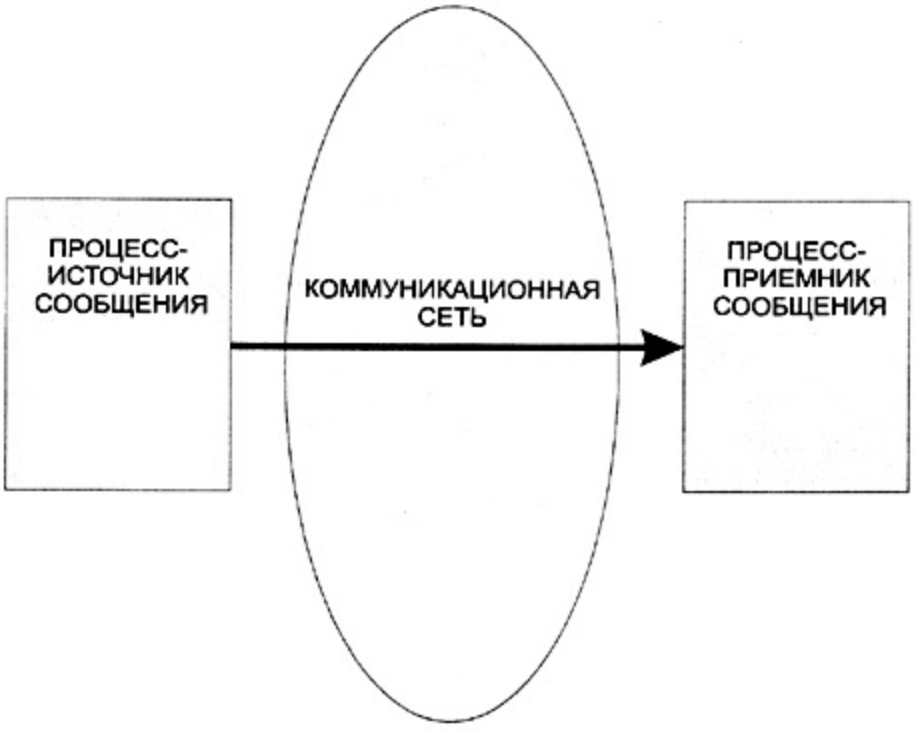


Рис. 4.1. *Двухточечный обмен*

Напомним, что во всех реализациях MPI, в том числе и в MPICH, гарантируется выполнение некоторых свойств двухточечного обмена. Одним из них является сохранение порядка сообщений, которые при двухточечном обмене не могут "обгонять" друг друга. Если, например, процесс с рангом 0 передает процессу с рангом 1 два сообщения: A и B, процесс 1 получит сначала сообщение A, а затем B. Другое важнейшее свойство — гарантированное выполнение обмена. Если один процесс посылает сообщение, а другой -запрос на его прием, то либо передача, либо прием будут считаться выполненными. При этом возможны три сценария обмена:

- второй процесс получает от первого адресованное ему сообщение;
- отправленное сообщение может быть получено третьим процессом, при этом фактически выполнена будет передача сообщения, а не его прием (сообщение прошло мимо адресата);
- второй процесс получает сообщение от третьего, тогда передача не может считаться выполненной, потому что адресат получил не то "письмо".

В двухточечном обмене следует соблюдать правило соответствия типов передаваемых и принимаемых данных. Это затрудняет обмен сообщениями между программами, написанными на разных языках программирования.

В предыдущей главе упоминалось о том, что есть четыре разновидности двухточечного обмена: синхронный, асинхронный, блокирующий и неблокирующий. В MPI имеются также четыре режима обмена, различающиеся условиями инициализации и завершения передачи сообщения:

- *стандартная передача* считается выполненной и завершается, как только сообщение отправлено, независимо от того, дошло оно до адресата или нет. В стандартном режиме передача сообщения может начинаться, даже если еще не начат его прием;
- *синхронная передача* отличается от стандартной тем, что она не завершается до тех пор, пока не будет завершен прием сообщения". Адресат, получив сообщение, посылает процессу, отправившему его, уведомление, которое должно быть получено отправителем для того, чтобы обмен считался выполненным. Операцию передачи уведомления иногда называют "рукопожатием";
- *буферизованная передача* завершается сразу же, сообщение копируется в системный буфер, где и ожидает своей очереди на пересылку. Завершается буферизованная передача независимо от того, выполнен прием сообщения или нет;
- *передача "по готовности" начинается* только в том случае, когда адресат инициализировал прием сообщения, а завершается сразу, независимо от того, принято сообщение или нет.

Каждый из этих четырех режимов имеется как в блокирующей, так и в неблокирующей формах.

В MPI приняты следующие соглашения об именах подпрограмм двухточечного обмена:

MPI_[I][R, S, B]Send

здесь префикс [I] (Immediate) обозначает неблокирующий режим. Один из префиксов [R, s, в] обозначает режим обмена, соответственно: по готовности, синхронный и буферизованный. Отсутствие префикса обозначает подпрограмму стандартного

обмена. Таким образом, имеется 8 разновидностей операции передачи сообщений.

Для подпрограмм приема:

MP1_[I]Recv

т. е. всего 2 разновидности приема. Подпрограмма MPi__irsend, например, выполняет передачу "по готовности" в неблокирующем режиме, MPI_Bsend -буферизованную передачу с блокировкой, а MPI_Recv выполняет блокирующий прием сообщений. Заметим, что подпрограмма приема любого типа может принять сообщения от любой подпрограммы передачи.

Блокирующие операции обмена

Блокирующие операции приостанавливают выполнение вызывающего процесса, заставляя процесс ожидать завершения передачи данных. Блокировка гарантирует выполнение действий в заданном порядке, но и создает условия для возникновения тупиковых ситуаций, когда оба процесса-участника обмена "точка-точка" блокируются одновременно.

Стандартный обмен

Сообщение, отправленное в стандартном режиме, может в течение некоторого времени "гулять" по коммуникационной сети параллельного компьютера, увеличивая тем самым ее загрузку. В MPI-программах рекомендуется соблюдать следующие правила:

- блокирующие операции обмена следует использовать осторожно, поскольку в этом случае возрастает вероятность тупиковых ситуаций;
- если для правильной работы процесса имеет значение последовательность приема сообщений, источник должен передавать новое сообщение, только убедившись, что предыдущее уже принято. В противном случае может нарушиться детерминизм выполнения программы;
- сообщения должны гарантированно и с достаточно большой частотой приниматься процессами, которым они посылаются, иначе коммуникационная сеть может переполниться, что приведет к резкому падению скорости ее работы. Это, в свою очередь, снизит производительность всей системы.

Основными подпрограммами двухточечного обмена сообщениями являются MPI_Send и MPI_Recv. MPI_send — это подпрограмма стандартной блокирующей передачи. Она блокирует выполнение процесса до завершения передачи сообщения (что, однако, не гарантирует завершения его приема):

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm)

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERR)

Здесь и далее в первой строке приводится прототип функции для языка С, а во второй — заголовок процедуры для языка FORTRAN. Входные параметры подпрограммы MPi_send:

- buf — адрес первого элемента в буфере передачи;
- count — количество элементов в буфере передачи;
- datatype — тип MPI каждого пересылаемого элемента;
- dest — ранг процесса-получателя сообщения. Ранг здесь — целое число от 0 до *п* — 1, где *п* — число процессов в области взаимодействия;
- tag — тег сообщения;
- comm — коммуникатор;
- ierr — код завершения.

По умолчанию реакцией системы на возникновение ошибки во время выполнения программы является ее останов, однако реакцию можно изменить с помощью вызова подпрограммы MPi_Errhandier_set. Следует иметь в виду, однако, что MPI не гарантирует нормального продолжения работы программы после возникновения ошибки.

Кроме ошибок, упоминавшихся в *главе* 5, при вызове подпрограмм обмена могут быть и такие:

- MPI__ERR_COMM — неправильно указан коммуникатор. Часто возникает при использовании "пустого" коммуникатора;
- MPI_ERR_COUNT — неправильное значение аргумента count (количество пересылаемых значений);
- MPI_ERR_TYPE — неправильное значение аргумента, задающего тип данных;
- MPI_ERR_TAG — неправильно указан тег сообщения;
- MPI_ERR_RANK — неправильно указан ранг источника или адресата сообщения;
- MPI_ERR_ARG — неправильный аргумент, ошибочное задание которого не попадает ни в один класс ошибок;
- MPI_ERR_REQUEST — неправильный запрос на выполнение операции.

Стандартная блокирующая передача начинается независимо от того, был ли зарегистрирован соответствующий прием, а завершается только после того, как сообщение принято системой и процесс-источник может вновь использовать буфер передачи. Сообщение может быть скопировано прямо в буфер приема, а может быть помещено во временный системный буфер, где и будет дожидаться вызова адресатом подпрограммы приема. В этом случае говорят о буферизации сообщения.

Она требует выполнения дополнительных операций копирования из памяти в память, а также выделения памяти для промежуточного буфера. В стандартном режиме MPI самостоятельно решает, надо ли буферизовать исходящие сообщения. Передача может завершиться еще до вызова соответствующей операции приема. С другой стороны, буфер может быть недоступен или MPI может решить не буферизовать исходящие сообщения по соображениям сохранения высокой производительности. В этом случае передача завершится только после того, как будет зарегистрирован соответствующий прием и данные будут переданы адресату.

Стандартный блокирующий прием выполняется подпрограммой:

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,

int tag, MPI_Comm comm, MPI_Status *status)

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERR)

Ее входные параметры:

- count -- максимальное количество элементов в буфере приема. Фактическое их количество можно определить с помощью подпрограммы MPI_Get_count;
- datatype — тип принимаемых данных. Напомним о необходимости соблюдения соответствия типов аргументов подпрограмм приема и передачи;
- source — ранг источника. Можно использовать специальное значение MPI_ANY_SOURCE, соответствующее произвольному значению ранга. В программировании идентификатор, отвечающий произвольному значению параметра, часто называют "джокером". Этот термин будем использовать и мы;
- tag — тег сообщения или "джокер" MPI_ANY_TAG, соответствующий произвольному значению тега;
- comm — коммуникатор. При указании коммуникатора "джокеры" использовать нельзя.

Следует иметь в виду, что при использовании значений MPI_ANY_SOURCE (любой источник) и MPI_ANY_TAG (любой тег) есть опасность приема сообщения, не предназначенного данному процессу.

Выходными параметрами являются:

- buf — начальный адрес буфера приема. Его размер должен быть достаточным, чтобы разместить принимаемое сообщение, иначе при выполнении приема произойдет сбой — возникнет ошибка переполнения;
- status — статус обмена.

Если сообщение меньше, чем буфер приема, изменяется содержимое лишь тех ячеек

памяти буфера, которые относятся к сообщению. Информация о длине принятого сообщения содержится в одном из полей статуса, но к этой информации у программиста нет прямого доступа (как к полю структуры или элементу массива). Размер полученного сообщения (count) можно определить с помощью вызова подпрограммы MPi_Get_count:

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERR)

Аргумент datatype должен соответствовать типу данных, указанному в операции обмена.

Подпрограмма MPI_Recv может принимать сообщения, отправленные в любом режиме. Имеется некоторая асимметрия между операциями приема и передачи. Она состоит в том, что прием может выполняться от произвольного процесса, а в операции передачи должен быть указан вполне определенный адрес. Приемник может использовать "джокеры" для источника и для тега. Процесс может отправить сообщение и самому себе, но следует учитывать, что в этом случае блокирующие операции могут привести к "тупику".

Программа, в которой используется двухточечный обмен, схематически представлена на рис. 4.2. Перед завершением программы и выполнением MPI_Finalize следует убедиться, что приняты все сообщения, отправленные ранее.

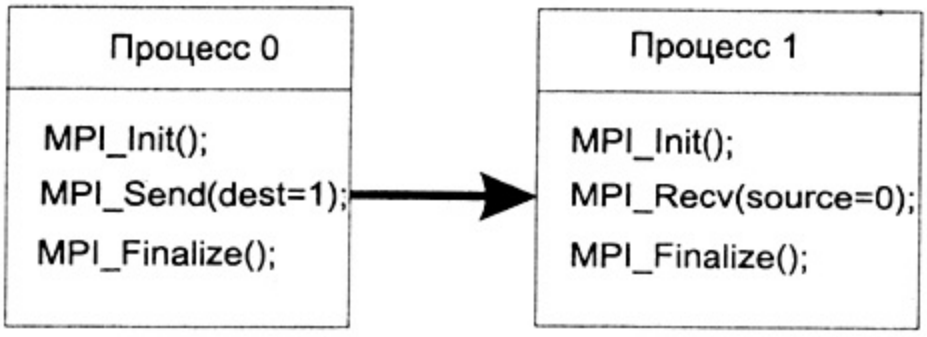


Рис. 4.2. *Схема простейшей программы с двухточечным обменом*

Если во втором процессе не предусмотрен прием отправленного ему сообщения, программу следует считать ошибочной (рис. 4.3).

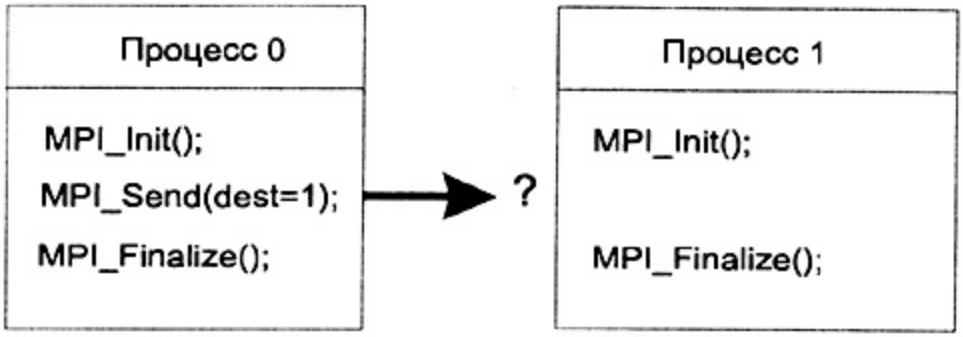


Рис. 4.3. *Схема ошибочной программы с двухточечным обменом*

Пример двухточечного обмена приведен в листинге 4.1. Это программа в духе знаменитой программы "Hello, world!" - - примера из классической книги Б. Кернигана и Д. Ритчи "The C Programming Language". Здесь используется SPMD-модель программирования.

Листинг 4.1. Пример 1 использования стандартного блокирующего обмена

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

{

int myid, numprocs; char message[24] ;

int myrank; MPI_Status status;

int TAG = 0; MPI_Init(&argc, &argv);

MPI_Comm_rank (MPI_COMM_WORLD, Smyrank) ;

if (myrank == 0) {

strcpy(message, "Hi, Parallel Programmer!");

MPI_Send(&message, 25, MPI_CHAR, 1, TAG, MPI_COMM_WORLD); }

else {

MPI_Recv(&message, 25, MPI_CHAR, 0,

TAG, MPI_COMM_WORLD, &status);
```

```
printf("received: %s\n", message); }

MPI_Finalize(); return 0; }
```

В данном примере процесс с рангом 0 передает сообщение процессу с рангом 1, используя для этого подпрограмму MPI_Send. При выполнении этой операции в памяти процесса-отправителя выделяется буфер передачи, который содержит переменную message. В буфере содержатся 24 символа. Процесс с рангом 1 принимает сообщение с помощью подпрограммы MPi_Recv. Сообщение сохраняется в буфере приема. Первые три аргумента подпрограммы приема определяют положение, размер и тип буфера приема. Результат выполнения этой программы:

```
# mpirun -np 2 a.out

received: Hi, Parallel Programmer!
```

В первой строке приведена командная строка запуска программы. Здесь и далее будем полагать, что имя исполняемого файла — a.out. Пример программы, в котором сообщениями обмениваются процессы с четными и нечетными рангами, дан в листинге 4.2. Предполагается, что значение size четно.

Листинг 4.2. Пример 2 использования стандартного блокирующего обмена

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

int myrank, size, message;

int TAG = 0;

MPI_Status status;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, Smyrank);

MPI_Comm_size(MPI_COMM_WORLD, &size);

message = myrank;

if((myrank % 2) == 0)

{
```

```
if((myrank + 1) != size) MPI_Send(&message, 1, MPI_

INT, myrank + 1, TAG, MPI_COMM_WORLD);

}

else

{

if(myrank != 0)

MPI_Recv(&message,- 1, MPI_INT, myrank — 1, TAG, MPI_COMM_WORLD, &status);

printf("received :%i\n", message);

}

MPI_Finalize();

return 0;

}
```

Результат выполнения этой программы в случае запуска 10 процессов выглядит так:

```
# mpirun -np 10 a.out received :0 received :2 received :6 received :4 received :8
```

Порядок вывода сообщений может быть другим, он меняется от случая к случаю.

Если стандартная передача не может быть выполнена из-за недостаточного объема приемного буфера, осуществление процесса блокируется до тех пор, пока не будет доступен буфер достаточного размера. Иногда это может оказаться удобным. Приведем пример. Пусть источник циклически посылает новые значения адресату и пусть они вырабатываются быстрее, чем потребитель может их принять. При использовании буферизованной передачи может произойти переполнение буфера приема. Для того чтобы его избежать, в программу придется включить дополнительную синхронизацию, а при использовании стандартной передачи такая синхронизация выполняется автоматически. Иногда недостаточный размер буфера может привести к тупиковой ситуации. Это иллюстрируется примерами, приведенными в листингах 4.3 и 4.4.

Листинг 4.3. Пример устойчиво работающей программы

```
PROGRAM MAIN_MPI
```

```
INCLUDE 'mpif.h'

INTEGER RANK, TAG, CNT, IERR, STATUS(MPI_STATUS_SIZE)

REAL SNDBUF1, SNDBUF2, RCVBUF

CNT = 1

TAG = 0

CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM__WORLD, RANK, IERR)

IF (RANK.EQ.O) THEN

SNDBUF1 = 3.14159

CALL MPI_SEND(SNDBUF1, CNT, MPI_REAL, 1,

TAG, MPI_COMM_WORLD, IERR}

PRINT *, "Process ", RANK, " send ", SNDBUF1

CALL MPI_RECV(RCVBUF, CNT, MPI_REAL, 1, TAG,

MPI_COMM_WORLD, STATUS, IERR)

PRINT *, "Process ", RANK, " received ", RCVBUF ELSE

CALL MPI_RECV(RCVBUF, CNT, MPI_REAL, 0, TAG,

MPI_COMM_WORLD, STATUS, IERR)

PRINT *, "Process ", RANK, " received ", RCVBUF

SNDBUF2 = 2.71828

CALL MPI_SEND(SNDBUF2, CNT, MPI_REAL, 0,

TAG, MPI_COMM_WORLD, IERR)

PRINT *, "Process ", RANK, " send ",, SNDBUF2 END IF

CALL MPI_FINALIZE(IERR) STOP END
```

Вывод этой программы приводится ниже:

mpirun -np 2 a.out Process 1 received 3.14159012

Process 1 send 2.71828008 Process 0 send 3.14159012

Process 0 received 2.71828008

Листинг 4.4. Пример программы, попадающей в "тупик"

```
PROGRAM MAIN_MPI INCLUDE 'mpif.h'
INTEGER RANK, TAG, CNT, IERR,
STATUS(MPI_STATUS_SIZE)
REAL SNDBUF, RCVBUF
TAG = 0
SNDBUF = 3.14159
CNT = 1
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_
COMM_WORLD, RANK, IERR)
IF (RANK.EQ.O) THEN '
CALL MPI_RECV(RCVBUF, CNT, MPI_
REAL, 1, TAG, MPI_COMM_WORLD, STATUS,
IERR) CALL MPI_SEND(SNDBUF, CNT, MPI_
REAL, 1, TAG, MPI_COMM_WORLD, IERR)
ELSE
CALL MPI_RECV(RCVBUF, CNT, MPI_REAL, 0,
TAG, MPI_COMM_WORLD, STATUS, IERR)
CALL MPI_SEND(SNDBUF, CNT, MPI_REAL, 0,
TAG, MPI_COMM_WORLD, IERR)
```

```
END IF

CALL MPI_FINALIZE(IERR)

STOP

END
```

Первая из этих программ работает нормально, а вторая попадает в "тупик" (поэтому и результат ее выполнения здесь не приводится — его просто нет). Во втором случае прием сообщения процессом с рангом 0 должен завершиться до передачи им своего сообщения. Этот прием может быть осуществлен, только если процессом с рангом 1 выполнена соответствующая передача, а она, в свою очередь, может начаться только после завершения процессом 1 приема сообщения от процесса 0.

Синхронный блокирующий обмен

При синхронном обмене адресат посылает источнику "квитанцию" - уведомление о завершении приема. Только после получения этого уведомления обмен считается завершенным и источник "знает", что его сообщение получено. Синхронная передача выполняется с помощью подпрограммы MPi_Ssend:

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm)

MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERR)
```

Ее параметры совпадают с параметрами подпрограммы Mpi_send.

Если процесс выполняет блокирующую синхронную передачу до того, как другой процесс попытается получить сообщение, он приостанавливается до приема сообщения адресатом. Синхронный обмен медленнее, чем стандартный, но он безопаснее, поскольку не дает коммуникационной сети переполниться не дошедшими до адресата, "потерянными в сети", сообщениями. Этим обеспечивается большая степень предсказуемости поведения параллельной программы. Благодаря отсутствию потерявшихся, "невидимых" сообщений, проще оказывается и отладка программы.

Передача сообщения в синхронном режиме может быть начата независимо от того, был ли зарегистрирован его прием, но она считается успешно выполненной только при наличии соответствующего приема. Завершение синхронной передачи означает не только то, что буфер передачи можно использовать заново, но и то, что адресат начал выполнение приема. Если и передача, и прием являются блокирующими операциями, обмен не завершится, пока оба процесса не начнут передачу и прием сообщения. Операция передачи в этом режиме нелокальна.

Буферизованный обмен

Передача сообщения в буферизованном режиме может быть начата независимо от того, зарегистрирован ли соответствующий прием. Источник копирует сообщение в буфер, а затем передает его в неблокирующем режиме так же, как в стандартном режиме. Эта операция локальна, поскольку ее выполнение не зависит от наличия соответствующего приема. Если объем буфера недостаточен, возникает ошибка. Выделение буфера и его размер контролируются программистом.

Буферизованная передача завершается сразу, поскольку сообщение немедленно копируется в буфер для последующей передачи. В отличие от стандартного обмена, в этом случае работа источника и адресата не синхронизована. Параметры подпрограммы буферизованного обмена MPI_Bsend:

int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm)

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERR)

такие же, как и у MPI_send.

При выполнении буферизованного обмена программист должен заранее создать буфер достаточного размера. Это делается с помощью вызова подпрограммы:

int MPI_Buffer_attach(void *buf, size) MPI_BUFFER_ATTACH(BUF, SIZE, IERR)

В результате вызова создается буфер buf размером size байтов. В программах на языке FORTRAN роль буфера может играть массив. Этот массив должен быть описан в программе, его не следует использовать для других целей (например, в качестве первого аргумента подпрограммы MPI_Bsend). За один раз к процессу может быть подключен только один буфер.

После завершения работы с буфером его необходимо отключить. Делается это с помощью вызова подпрограммы:

int MPI_Buffer_detach(void *buf, int *size)

MPI BUFFER DETACH(BUF, SIZE, IERR)

В результате выполнения вызова возвращается адрес (buf) и размер отключаемого буфера (size). Эта операция блокирует работу процесса до тех пор, пока все сообщения, находящиеся в буфере, не будут обработаны. Благодаря этому, вызов данной подпрограммы можно использовать для форсированной передачи сообщений. После завершения вызова можно вновь использовать память, которую занимал

буфер, однако следует помнить, что в языке С данный вызов не освобождает автоматически память, отведенную для буфера. Пример использования подпрограмм подключения и отключения буфера приведен в листинге 4.5.

Листинг 4.5. Пример использования буферизованного обмена

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

{

int *buffer;

int myrank;

MPI_Status status;

int buffsize = 1;

int TAG = 0;

MPI_Init(&argc, Sargv);

MPI_Comm_rank(MPI_COMM_WORLD, anyrank);

if (myrank == 0}

{

buffer = (int *) malloc(buffsize +

MPI_BSEND_OVERHEAD);

MPI_Buffer_attach(buffer, buffsize +

MPI_BSEND_OVERHEAD); buffer = (int *) 10;

MPI_Bsend(&buffer, buffsize, MPI_INT, 1, TAG,

MPI_COMM_WORLD); MPI_Buffer_detach(&buffer, Sbuffsize);

} else

{
```

```
MPI_Recv(&buffer, buffsize, MPI_INT, 0,

TAG, MPI_COm_WORLD, Sstatus) ;

printf("received: %i\n", buffer);

}

MPI_Finalize() ;

return 0; }
```

Обратите внимание на то, что размер буфера должен превосходить размер сообщения на величину MPI_BSEND_OVERHEAD. Это дополнительное пространство используется подпрограммой буферизованной передачи для своих целей.

Если перед выполнением операции буферизованного обмена не выделен буфер, MP1 ведет себя так, как если бы с процессом был связан буфер нулевого размера. Работа с таким буфером обычно завершается сбоем программы.

Буферизованный обмен рекомендуется использовать в тех ситуациях, когда программисту требуется больший контроль над распределением памяти. Этот режим удобен и для отладки, поскольку причину переполнения буфера определить легче, чем причину тупика.

Обмен "по готовности"

Передача "по готовности" должна начинаться, если уже зарегистрирован соответствующий прием. Завершается она сразу же. Если прием не зарегистрирован, результат выполнения операции не определен. Завершение передачи не зависит от того, вызвана ли другим процессом подпрограмма приема данного сообщения или нет, оно означает только, что буфер передачи можно использовать вновь. Сообщение просто выбрасывается в коммуникационную сеть в надежде, что адресат его получит. Следует иметь в виду, что эта надежда может и не сбыться. "Правила игры" здесь такие же, как и в операциях стандартного или синхронного обмена. В "правильной" программе передача "по готовности" может быть заменена стандартной передачей, результат выполнения и поведение программы при этом не должны измениться. Измениться может только скорость ее выполнения. Передача "по готовности" выполняется с помощью подпрограммы MPI_Rsend:

```
int MPI_Rsend(void *buf, int count,

MPI_Datatype datatype, int dest,

int tag, MPI_Comm сошп)
```

MPI_RSEND(BUF, COUNT, DATATYPE,

DEST, TAG, COMM, IERR)

Параметры у нее те же, что и у подпрограммы MPI_send.

Обмен "по готовности" может увеличить производительность программы, поскольку здесь не используются этапы установки межпроцессных связей, а также буферизация. Все это — операции, требующие времени. С другой стороны, обмен "по готовности" потенциально опасен, кроме того, он усложняет отладку, поэтому его рекомендуется использовать только в том случае, когда правильная работа программы гарантируется ее логической структурой (рис. 4.4), а выигрыша в быстродействии надо добиться любой ценой.

Подпрограммы-пробники

Получить информацию о сообщении еще до его помещения в буфер приема можно с помощью подпрограмм-пробников (или зондирующих подпрограмм) MPi_Probe и MPi_iProbe. На основании полученной информации принимается решение о дальнейших действиях. Можно, например, выделить

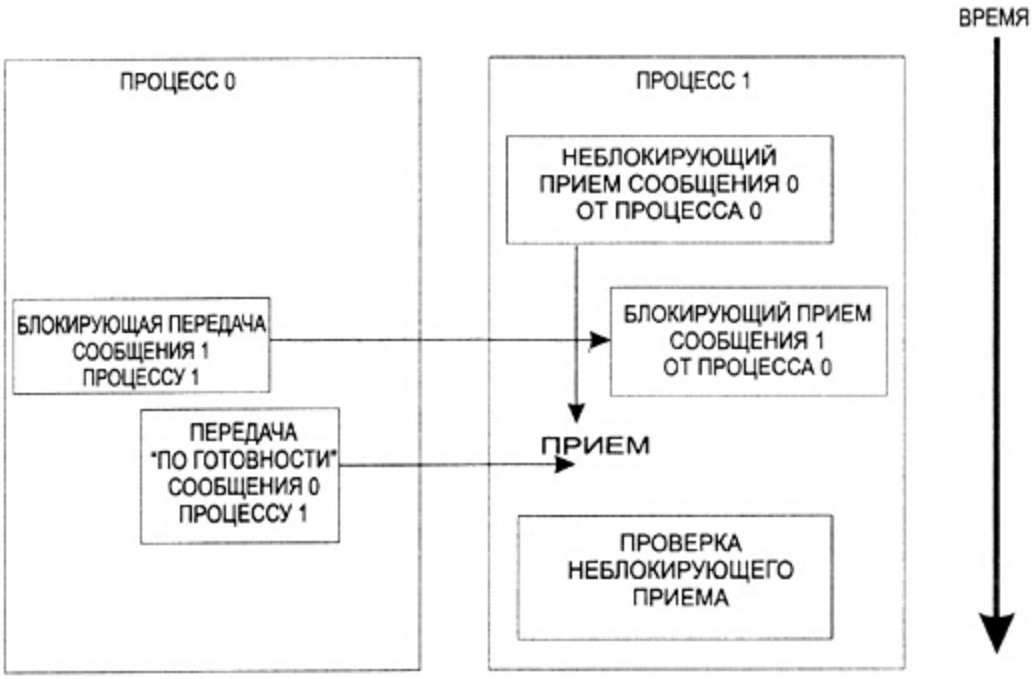


Рис. 4.4. *"Безопасная" структура программы* с использованием обмена "по готовности"

буфер приема достаточного размера, определив длину сообщения с помощью подпрограммы MPI_Get_count, которая дает доступ к одному из полей статуса. Если после вызова MPI_Probe следует вызов MPI_Recv с такими же значениями аргументов, что и MPI_probe, он поместит в буфере приема то же самое сообщение, информация о котором была получена MPI_ProЬe.

Подпрограмма MPI_probe выполняет блокирующую проверку доставки сообщения:

int MPI_Probe(int source, int tag, MPI_Comm comm,

MPI_Status *status) MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERR)

Ее входные параметры:

- source — ранг источника или "джокер";
- tag — значение тега или "джокер";
- comm — коммуникатор.

Выходным параметром является статус (status), который и содержит необходимую информацию.

Неблокирующая проверка сообщения выполняется подпрограммой

MPI_Iprobe:

int MPI Iprobe(int source, int tag, MPI_Comm comm, int *flag,

MPI_Status *status)

MPI IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERR)

Входные параметры те же, что и у подпрограммы MPI probe, а выходные: flag — флаг и status — статус. Если сообщение уже поступило и может быть принято, возвращается значение флага "истина". Проверка приема может выполняться с помощью MPI_Iprobe неоднократно. Не обязательно выполнять прием сообщения сразу же после его поступления. Пример использования блокирующего зондирования приведен в листинге 4.6.

Листинг 4.6. Пример использования подпрограмм блокирующего зондирования

```
PROGRAM MAIN_MPI

INCLUDE 'mpif.h'

INTEGER RANK, I, K, IERR, TAG, DEST, STATUS(MPI_STATUS_SIZE)

REAL X

TAG = 0

DEST = 2
```

```
CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)

IF (RANK.EQ.O) THEN

I = 2002

CALL MPI_SEND(I, 1, MPI_INTEGER, DEST,

TAG, MPI_COMM_WORLD, IERR} ELSE

IF(RANK.EQ.I) THEN X = 3.14159

CALL MPI__SEND(X, 1, MPI_REAL, DEST,

TAG, MPI_COMM_WORLD, IERR) ELSE DO К = 1, 2

CALL MPI_PROBE(MPI__ANY_SOURCE,

TAG, MPI_COMM_WORLD, STATUS, IERR)

IF (STATUS(MPI_SOURCE).EQ.0) THEN

CALL MPI_RECV(I, 1, MPI_INTEGER, 0, TAG,

MPI_COMM_WORLD, STATUS, IERR)

PRINT *, "Received ", I, " from 0" ELSE

CALL MPI_RECV(X, 1, MPI_REAL, 1, TAG,

MPI_COMM_WORLD, STATUS, IERR)

PRINT *, "Received ", X, " from 1"

END IF END DO END IF

CALL MPI_FINALIZE(IERR) STOP END
```

В этом примере проблема заключается в том, что процессы с рангами 0 и 1 передают процессу 2 сообщения, содержащие данные разных типов. Порядок приема этих сообщений не определен, поэтому заранее неизвестно,

данные какого типа должны быть приняты при первом вызове подпрограммы MPi_Recv, а какие - при втором. Благодаря вызову подпрограммы MPi_probe, каждое

сообщение принимается с использованием правильного типа данных.

В примере из листинга 4.7 подпрограммы-пробники принимают сообщения от неизвестного источника, которые к тому же содержат неизвестное количество элементов целого типа. В этом случае вместо ранга источника и тега сообщения используются "джокеры". Сначала с помощью вызова подпрограммы MPi_Probe фиксируется поступление (но не прием!) сообщения. Затем определяется источник сообщения, с помощью вызова MPi_Get_count определяется его длина, выделяется буфер подходящего размера и выполняется прием сообщения.

Листинг 4.7. Пример использования зондирующих подпрограмм

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

{

int myid, numprocs, **buf, source, i;

int message[3] = {0, 1, 2};

int myrank, data = 2002, count, TAG = 0;

MPI_Status status;

MPI_Init(&argc, &argv);

MPI_Coim_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0)

{

MPI_Send(&data, 1, MPI_INT, 2, TAG,

MPI_COMM_WORLD); } else if (myrank == 1) {

MPI_Send(&message, 3, MPI_INT, 2,

TAG, MPI_COMM_WORLD); }

else {
```

```
MPI_Probe(MPI_ANY_SOURCE, 0, MPI_

COMM_WORLD, &status); source = status.MPI_

SOURCE; MPI_Get_count(&status, MPI_INT, &count);

for (i = 0; i < count; i++){ buf[i] = (int *)malloc(count*sizeof(int));

}

MPI_Recv(Sbuf[0], count, MPI_INT, source,

TAG, MPI_COMM_WORLD, Sstatus);

for (i = 0; i < count; i++){

printf ("received: %d\n", buf[i]);

MPI_Finalize() ; return 0;
```

При запуске этой программы должны создаваться три процесса:

```
# mpirun -np 3 a.out
```

Процесс с рангом 0 передает процессу 2 сообщение, содержащее одно значение (переменная data), а процесс с рангом 1 передает тому же процессу три элемента данных в массиве message. Порядок поступления этих сообщений не определен и первым может прийти как сообщение от процесса 0, так и сообщение от процесса 1. Для того чтобы правильно организовать прием сообщения, необходимо заранее узнать его длину и, возможно, источник. Это и делают подпрограммы Mpi_probe и MPi_Get_count. Заметим, что в данном примере мы сознательно нарушили ранее данные рекомендации, — ведь из двух отправленных сообщений принято будет лишь одно, а таких ситуаций следует избегать.

Совместные прием и передача

Операции приемопередачи объединяют в едином вызове передачу сообщения одному процессу и прием сообщения от другого процесса. Данный вид обменов может оказаться полезным при выполнении сложных схем обмена сообщениями, например, по цепи процессов (рис. 4.5). Если для сдвига применяются блокирующие операции передачи и приема, определить их порядок следует правильно, иначе взаимные зависимости между процессами могут привести к тупиковым ситуациям. Четные процессы, например, могут работать вначале на передачу, а нечетные на прием. Затем они меняются ролями. При использовании операций приемопередачи сама система заботится о соблюдении правильной последовательности обменов.

Подпрограммы приемопередачи могут взаимодействовать с обычными подпрограммами обмена и подпрограммами зондирования.

Подпрограмма MPI_sendrecv выполняет прием и передачу данных с блокировкой:

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,

int dest, int sendtag, void *recvbuf, int recvcount,

MPI_Datatype recvtype, int source, int recvtag, MPI_Coinm comm,

MPI_Status *status)

MPI_SENDRECV(SENDBUF, SENDCOUNT,

SENDTYPE, DEST, SENDTAG, RECVBUF,

RECVCOUNT, RECVTYPE, SOURCE,

RECVTAG, COMM, STATUS, IERR)

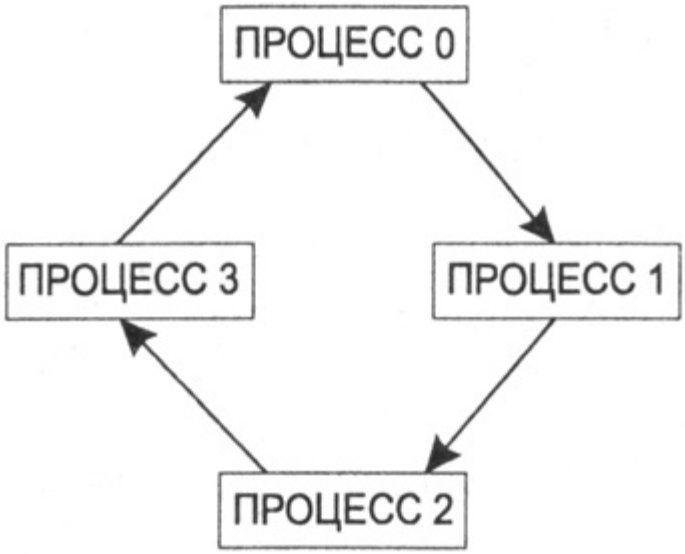


Рис. 4.5. *Обмен сообщениями по цепи процессов, замкнутой в кольцо*

Ее входные параметры:

- sendbuf — начальный адрес буфера передачи;
- sendcount — количество передаваемых элементов;
- sendtype — тип передаваемых элементов;
- dest — ранг адресата;
- sendtag — тег передаваемого сообщения;
- recvbuf — начальный адрес буфера приема;

- recvcount — количество элементов в буфере приема;
- recvtype — тип элементов в буфере приема;
- source — ранг источника;
- recvtag — тег принимаемого сообщения;
- comm — коммуникатор.

Выходные параметры: recvbuf — начальный адрес буфера приема и status -статус операции приема. И прием, и передача используют один и тот же коммуникатор. Буферы передачи и приема не должны пересекаться, у них может быть разный размер, типы пересылаемых и принимаемых данных также могут различаться.

Подпрограмма MPI_sendrecv_replace отправляет и принимает сообщение в блокирующем режиме, используя общий буфер для передачи и для приема:

int MPI_Sendrecv_replace(void *buf, int count,

MPI_Datatype datatype, int dest, int sendtag,

int source, int recvtag, MPI_Comm comm, MPI_Status *status)

MPI_SENDRECV_REPLACE(BUF, COUNT,

DATATYPE, DEST, SENDTAG, SOURCE,

RECVTAG, COMM, STATUS, IERR)

Входные параметры:

- count — количество отправляемых данных и емкость буфера приема;
- datatype — тип данных в буфере приема и передачи;
- dest — ранг адресата;
- sendtag — тег передаваемого сообщения;
- source — ранг источника;
- recvtag — тег принимаемого сообщения;
- comm — коммуникатор.

Выходные параметры: buf — начальный адрес буфера приема и передачи и статус (status). Принимаемое сообщение не должно превышать по размеру отправляемое сообщение, а передаваемые и принимаемые данные должны быть одного типа. Последовательность приема и передачи выбирается системой автоматически.

Мы уже говорили о том, что большинство реализаций MPI гарантируют соблюдение определенных свойств двухточечного обмена. Важнейшим из них является сохранение порядка сообщений при приеме. Данное свойство обеспечивает детерминированное поведение параллельной программы в случае, когда процессы выполняются в одном

потоке, а операции приема не используют "джокер" MPI_ANY_SOURCE. Нарушить детерминированное поведение

Программы могут вызывать подпрограмм MPI_Cancel и MPI_Waitany.

Пример программы с передачей и приемом сообщений, порядок которых сохраняется при приеме, приведен в листинге 4.8.

Листинг 4.8. Пример передачи и приема сообщений с сохранением порядка

```
PROGRAM MAIN_MPI

INCLUDE 'mpif.h'

INTEGER RANK, TAG, CNT, IBUF, IERR, STATUS(MPI_STATUS_SIZE)

REAL SNDBUF1, SNDBUF2, RCVBUF

PARAMETER(BUFSIZE = 4 + MPI_BSEND_OVERHEAD)

BYTE BUFF(BUFSIZE)

CNT = 1

TAG = 0

CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)

IF (RANK.EQ.0) THEN

SNDBUF1 = 3.14159

CALL MPI_BUFFER_ATTACH(BUFF, BUFSIZE, IERR)

CALL MPI_BSEND(SNDBUF1, CNT, MPI_REAL,

1, TAG, MPI_COMM_WORLD, IERR)

PRINT *, "Process ", RANK, " send ", SNDBUF1

SNDBUF2 = 2.71828

CALL MPI_BSEND(SNDBUF2, CNT, MPI_REAL,

1, TAG, MPI_COMM_WORLD, IERR)
```

```
PRINT *, "Process ", RANK, " send ", SNDBUF2

CALL MPI BUFFER ATTACH(BUFF, IBUFSIZE, IERR)

ELSE

CALL MPI_RECV(RCVBUF, CNT, MPI_REAL, 0,

TAG, MPI_COMM_WORLD, STATUS, IERR)

PRINT *, "Process ", RANK, " received ", RCVBUF

CALL MPI_RECV(RCVBUF, CNT, MPI_REAL, 0,

TAG, MPI_COMM_WORLD, STATUS, IERR)

PRINT *,, "Process ", RANK, " received ", RCVBUF END IF

CALL MPI_FINALIZE(IERR) STOP END
```

Сообщение, отправленное первым, должно быть получено при первом вызове подпрограммы приема, а второе — при втором.

Следующее свойство двухточечного обмена заключается в том, что, если два процесса инициализировали передачу и прием сообщения, по крайней мере одна из этих операций будет выполнена. Передача будет выполнена, если только адресат не примет другое сообщение. Прием будет выполнен, если сообщение не будет перехвачено другим процессом, выполняющим обмен с тем же источником, как в примере из листинга 4.9.

Листинг 4.9. Пример перехвата сообщения

```
PROGRAM MAIN_MPI

INCLUDE 'mpif.h'

INTEGER RANK, TAG1, TAG2, CNT,

IERR, STATUS(MPI_STATUS_SIZE)

REAL SNDBUF1, SNDBUF2, RCVBUF1, RCVBUF2

PARAMETER(BUFSIZE = 4 +

MPI_BSEND_OVERHEAD)
```

```
BYTE BUFF(BUFSIZE)
CNT = 1
TAG1 = 0
TAG2 = 1
SNDBUF1 = 3.14159
SNDBUF2 = 2.71812
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
IF (RANK.EQ.0) THEN
CALL MPI_BUFFER_ATTACH(BUFF, BUFSIZE, IERR)
CALL MPI_BSEND(SNDBUF1, CNT,
MPI_REAL, 1, TAG1, MPI_COMM_WORLD, IERR)
PRINT *, "PROCESS ", RANK, " SEND ", SNDBUF1
CALL MPI_BUFFER_DETACH(BUFF, IBUFSIZE, IERR)
CALL MPI_SSEND(SNDBUF2, CNT,
MPI_REAL, 1, TAG2, MPI_COMM_WORLD, IERR)
PRINT *, "PROCESS ", RANK, " SEND ", SNDBUF2
ELSE
CALL MPI_RECV(RCVBUF1, CNT,
MPI_REAL, 0, TAG2, MPI_COMM_WORLD, STATUS,
IERR)
PRINT *, "PROCESS ", RANK, " RECEIVED ", RCVBUF1
CALL MPI_RECV(RCVBUF2, CNT, MPI_
REAL, 0, TAG1, MPI_COMM_WORLD, STATUS, IERR)
```

```
PRINT *, "PROCESS ", RANK, " RECEIVED ", RCVBUF2

END IF

CALL MPI_FINALIZE(IERR)

STOP

END
```

Здесь оба процесса вызывают свою первую подпрограмму передачи. Первая передача процесса с рангом 0 происходит в режиме буферизации, поэтому она будет выполнена независимо от состояния процесса с рангом 1. Запроса на прием нет, поэтому сообщение копируется в буфер, затем вызывается вторая подпрограмма передачи. В этот момент образуется пара соответствующих друг другу операций передачи и приема. Обе операции будут выполнены. Процесс с рангом 1 вызывает вторую операцию приема, которая принимает буферизованное сообщение:

```
# mpirun -np 2 a.out

Process 0 send 3.14159012

Process 0 send 2.71828004

Process 1 received 3.14159012

Process 1 received 2.71828004
```

Если бы обе передачи выполнялись, например, в синхронном блокирующем режиме, программа при запуске "зависла" бы (проверьте!). Использование буферизованной передачи позволяет избежать "тупика".

MPI не гарантирует "справедливой" обработки обменов. Приведем пример. Пусть процессом 0 процессу 1 отправлено сообщение. Может оказаться, что адресат осуществляет периодически повторяющиеся запросы на прием, тем не менее, не исключено, что сообщение от процесса 0 никогда не будет принято, потому что каждый раз раньше него будет приходить другое сообщение, отправленное другим процессом. Программист должен избегать подобных ситуаций.

Следует учитывать и ограниченность ресурсов компьютера. Любое сообщение, находящееся в коммуникационной сети, использует системные ресурсы. Эти ресурсы ограничены и при неблагоприятном стечении обстоятельств их может оказаться недостаточно. Таких ситуаций также следует избегать. Операция буферизованной передачи, которая не может быть выполнена из-за недостаточного объема буфера, может привести к аварийному останову программы.

Неблокирующие операции обмена

Для того чтобы избежать простоев во время выполнения обмена, используют неблокирующие операции. Вызов подпрограммы неблокирующей передачи инициирует, но не завершает ее. Завершиться выполнение подпрограммы может еще до того, как сообщение будет скопировано в буфер передачи. Применение неблокирующих операций улучшает производительность программы, поскольку в этом случае допускается перекрытие (т. е. одновременное выполнение) вычислений и обменов. Передача данных из буфера или их считывание может происходить одновременно с выполнением процессом другой работы. Для завершения обмена требуется вызов дополнительной процедуры, которая проверяет, скопированы ли данные в буфер передачи. Несмотря на то, что при неблокирующем обмене возвращение из подпрограммы обмена происходит сразу, запись в буфер или считывание из него после этого производить нельзя, ведь сообщение может быть еще не отправлено или не получено и работа с буфером может "испортить" его содержимое. Таким образом, неблокирующий обмен выполняется в два этапа:

1. Инициализация обмена.

2. Проверка завершения обмена.

Разделение этих шагов делает необходимым маркировку каждой операции обмена, что позволяет целенаправленно выполнять проверки завершения соответствующих операций. Для маркировки в неблокирующих операциях используются *идентификаторы операций обмена* (requests).

Неблокирующая передача может выполняться в тех же четырех режимах, что и блокирующая: стандартном, буферизованном, синхронном и "по готовности". Передача в каждом из них может быть начата независимо от того, был ли зарегистрирован соответствующий прием. Во всех случаях, операция начала передачи локальна, она завершается сразу же и независимо от состояния других процессов.

Инициализация неблокирующего обмена

Инициализация неблокирующей стандартной передачи выполняется с помощью подпрограммы MPI_Isend:

int MPI_Isend(void *buf, int count,

MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm, MPI_Request *request)

MPI_ISEND(BUF, COUNT, DATATYPE,

DEST, TAG, COMM, REQUEST, IERR)

Ее входные параметры аналогичны аргументам подпрограммы MPi_send, а выходным является параметр request — идентификатор операции.

Подпрограмма MPI_issend инициализирует неблокирующую синхронную передачу данных:

int MPI_Issend(void *buf, int count,

MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm,

MPI_Request ^request)

MPI_ISSEND(BUF, COUNT, DATATYPE,

DEST, TAG, COMM, REQUEST, IERR)

Ее параметры совпадают с параметрами подпрограммы MPI_isend.

Неблокирующая буферизованная передача сообщения выполняется подпрограммой MPI_Ibsend:

int MPI_Ibsend(void *buf, int count,

MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm, MPI_Request *request)

MPI_IBSEND(BUF, COUNT,

DATATYPE, DEST, TAG, COMM, REQUEST, IERR)

Неблокирующая передача "по готовности" выполняется подпрограммой

MPI_Irsend:

int MPI_Irsend(void* buf, int count,

MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm, MPI_request *request)

MPI_IRSEND(BUF, COUNT, DATATYPE,

DEST, TAG, COMM, REQUEST, IERR)

Параметры всех подпрограмм неблокирующей передачи совпадают. Подпрограмма MPi_irecv начинает неблокирующий прием:

int MPI_Irecv(void *buf, int count,

MPI_Datatype datatype, int source,

int tag, MPI_Comm comm, MPI_Request *request)

MPI_IRECV(BUF, COUNT, DATATYPE,

SOURCE, TAG, COMM, REQUEST, IERR)

Назначение аргументов здесь такое же, как и в предыдущих подпрограммах, за исключением того, что указывается ранг не адресата, а источника сообщения (source).

Вызовы подпрограмм неблокирующего обмена формируют запрос на выполнение операции обмена и связывают его с идентификатором операции request. Запрос идентифицирует различные свойства операции обмена, такие как режим, характеристики буфера обмена, контекст, тег и ранг, используемые при обмене. Кроме того, запрос содержит информацию о состоянии ожидающих обработки операций обмена и может быть использован для получения информации о состоянии обмена или для ожидания его завершения *(см. далее в этой главе).*

Вызов подпрограммы неблокирующей передачи означает, что система может начинать копирование данных из буфера. Источник не должен обращаться к буферу передачи во время выполнения неблокирующей передачи. Вызов неблокирующего приема означает, что система может начинать запись данных в буфер приема. Источник во время передачи и адресат во время приема не должны обращаться к буферу. Неблокирующая передача может быть принята подпрограммой блокирующего приема и наоборот.

Неблокирующий обмен можно использовать не только для увеличения скорости выполнения программы, но и в тех ситуациях, где блокирующий обмен может привести к "тупику". Избежать в этом случае "тупика" можно, используя и подпрограмму блокирующего обмена MPi_Sendrecv, но блокирующий обмен замедляет выполнение программы, поскольку его скорость определяется скоростью работы коммуникационной сети, а эта скорость, как правило, не очень велика.

И немного о терминах. Пустым называют запрос с идентификатором MPI_REQUEST_NULL. Отложенный запрос на выполнение операции обмена называют неактивным, если он не связан ни с каким исходящим сообщением. Пустым называют

статус, поле тега которого принимает значение MPI_ANY_TAG, поле источника сообщения - MPI_ANY_SOURCE, а вызовы подпрограмм MPI_Get_count и MPI_Get_eiements возвращают нулевое значение аргумента count.

Проверка выполнения обмена

Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью вызова подпрограмм ожидания, блокирующих работу процесса до завершения операции, или неблокирующих подпрограмм проверки, возвращающих логическое значение "истина", если операция выполнена.

Подпрограмма MPI_wait блокирует работу процесса до завершения приема или передачи сообщения:

int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERR)

Входной параметр request — идентификатор операции обмена, а выходной — статус (status). В аргументе status возвращается информация о выполненной операции. Значение статуса для операции передачи сообщения можно получить вызовом подпрограммы MPi_Test_canceiied. Можно вызвать MPi_wait с пустым или неактивным аргументом request. В этом случае операция завершается сразу же с пустым статусом.

Успешное выполнение подпрограммы MPi_wait после вызова MPI_Ibsend подразумевает, что буфер передачи можно использовать вновь, т. е. пересылаемые данные отправлены или скопированы в буфер, выделенный при вызове подпрограммы MPI_Buffer_attach. В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить подпрограмму MPI_Cancel, которая освобождает память, выделенную подсистеме коммуникаций.

Подпрограмма MP_Test выполняет неблокирующую проверку завершения приема или передачи сообщения:

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

MPI_TEST(REQUEST, FLAG, STATUS, IERR)

Ее входным параметром является идентификатор операции обмена request. Выходные параметры: flag - "истина", если операция, заданная идентификатором request, выполнена, и status — статус выполненной операции.

Если при вызове MPI_Test используется пустой или неактивный аргумент request,

операция возвращает значение флага "истина" и пустой статус. Функции MPI_Wait и MPI_Test можно использовать для завершения операций приема и передачи.

В том случае, когда сразу несколько процессов обмениваются сообщениями, можно использовать проверки, которые применяются одновременно к нескольким обменам. Есть три типа таких проверок:

- проверка завершения всех обменов;
- проверка завершения любого обмена из нескольких;
- проверка завершения заданного обмена из нескольких.

Каждая из этих проверок, в свою очередь, имеет две разновидности — "ожидание" и "проверка".

Проверка завершения всех обменов выполняется подпрограммой MPI_Waitall, которая блокирует выполнение процесса до тех пор, пока все операции обмена, связанные с активными запросами в массиве requests, не будут выполнены, и возвращает статус этих операций. Статус обменов содержится в массиве statuses:

int MPI_Waitall(int count, MPI_Request requests[],

MPI_Status statuses![]) MPI_WAITALL

(COUNT, REQUESTS, STATUSES, IERR)

Здесь count — количество запросов на обмен (размер массивов requests И statuses).

В результате выполнения подпрограммы MPI_waitall запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение MPI_REQUEST_NULL. Список может содержать пустые или неактивные запросы. Для каждого из них устанавливается пустое значение статуса.

В случае неуспешного выполнения одной или более операций обмена подпрограмма MPI_Waitall возвращает код ошибки MPI_ERR_IN_STATUS и присваивает полю ошибки статуса значение кода ошибки соответствующей операции. Если операция выполнена успешно, полю присваивается значение MPI_SUCCESS, а если не выполнена, но и не было ошибки — значение MPI_ERR_PENDING. Последний случай соответствует наличию запросов на выполнение операции обмена, ожидающих обработки.

Неблокирующая проверка выполняется с помощью вызова подпрограммы MPI_Testall. Он возвращает значение флага (flag) "истина", если все обмены, связанные с активными запросами в массиве requests, выполнены. Если завершены не все обмены, флагу присваивается значение "ложь", а массив statuses не определен:

int MPI_Testall(int count, MPI_Request requests[], int *flag,

MPI_Status statuses[])

MPI_TESTALL(COUNT, REQUESTS, FLAG, STATUSES, IERR)

Здесь count — количество запросов.

Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена. Если запрос был сформирован операцией неблокирующего обмена, он аннулируется, а его элементу массива присваивается значение MPI_REQUEST_NULL. Каждому статусу, соответствующему пустому или неактивному запросу, присваивается пустое значение.

Проверки завершения любого числа обменов выполняются с помощью подпрограмм MPI_waitany и MPI_Testany. Первый из этих вариантов — блокирующий. Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (requests) не будет завершен. Индекс соответствующего элемента в массиве requests возвращается в аргументе index, а статус — в аргументе status:

int MPI_Waitany(int count, MPI_Request requests[], int *index,

MPI_Status *status)

MPI_WAITANY(COUNT, REQUESTS, INDEX, STATUS, IERR)

Входные параметры: requests и count — количество элементов в массиве requests, а выходные: status и index — индекс запроса (в языке C это целое число от 0 до count - 1, а в языке FORTRAN от 1 до count).

Если запрос на выполнение операции был сформирован неблокирующей операцией обмена, он аннулируется и ему присваивается значение MPI_REQUEST_NULL. Массив запросов может содержать пустые или неактивные запросы. Если в списке вообще нет активных запросов или он пуст, вызовы завершаются сразу со значением индекса MPI_UNDEFINED и пустым статусом.

Подпрограмма MPI_Testany проверяет выполнение любого ранее инициализированного обмена:

int MPI_Testany(int count, MPI_Request requests[], int *index, int *flag,

MPI_Status *status)

MPI_TESTANY(COUNT, REQUESTS,

INDEX, FLAG, STATUS, IERR)

Смысл и назначение параметров этой подпрограммы те же, что и для подпрограммы MPI_Waitany, но есть дополнительный аргумент flag, который принимает значение "истина", если одна из операций завершена. Блокирующая подпрограмма MPI_Waitany и неблокирующая MPijrestany взаимозаменяемы, впрочем, как и другие аналогичные пары.

Подпрограммы MPI_Waitsome и MPI_Testsome действуют аналогично подпрограммам MPI_Waitany и MPijrestany, кроме случая, когда завершается более одного обмена. В подпрограммах MPI_waitany и MPijrestany обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для MPI_waitsome и MPI_Testsome статус возвращается для всех завершенных обменов. Эти подпрограммы можно использовать для определения, сколько обменов завершено:

int MPI_Waitsome (int incount, MPI_Request requests[], int *outcount,

int indices[], MPI_Status statuses[])

MPI_WAITSOME(INCOUNT, REQUESTS,

OUTCOUNT, INDICES, STATUSES, IERR)

Здесь incount — количество запросов. В outcount возвращается количество выполненных запросов из массива requests, а в первых outcount элементах массива indices возвращаются индексы этих операций. В первых outcount элементах массива statuses возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру outcount присваивается значение MPI_UNDEFINED.

Подпрограмма MPi_Testsome — это неблокирующая проверка:

int MPijTestsome(int incount, MPI_Request requests[], int *outcount,

int indices[], MPI_Status statuses[])

MPIJTESTSOME(INCOUNT, REQUESTS,

OUTCOUNT, INDICES, STATUSES, IERR)

У нее такие же параметры, как и у подпрограммы Mpi_waitsome. Эффективность подпрограммы MPijrestsome выше, чем у MPijrestany, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.

Отложенные обмены

Достаточно часто приходится сталкиваться с ситуацией, когда обмены с одинаковыми .параметрами выполняются повторно, например, в цикле. В этом случае можно объединить аргументы подпрограмм обмена в один отложенный запрос, который затем повторно используется для инициализации и выполнения обмена сообщениями. Отложенный запрос на выполнение неблокирующей операции' обмена позволяет минимизировать накладные расходы на организацию связи между процессором и контроллером связи.

Отложенные запросы на обмен объединяют такие сведения об операциях обмена, как адрес буфера, количество пересылаемых элементов данных, их тип, ранг адресата, тег сообщения и коммуникатор.

Для создания отложенного запроса на обмен используются четыре подпрограммы. Сам обмен они не выполняют. Запрос для стандартной передачи создается при вызове подпрограммы MPi_Send_init:

int MPI_Send_init(void *buf, int count,

MPI_Datatype datatype, int dest,

int tag, MPI_Comm comm, MPI_Request *request)

MPI_SEND_INIT(BUF, COUNT, DATATYPE,

DEST, TAG, COMM, REQUEST, IERR)

Ее входные параметры:

- buf — адрес буфера передачи;
- count — количество элементов;
- datatype — тип элементов;
- dest — ранг адресата;
- tag — тег сообщения;
- comm — коммуникатор.

Выходным параметром является запрос на выполнение операции обмена (request).

Отложенный запрос может быть сформирован для всех режимов обмена.

Для этого используются подпрограммы MPI_Bsend_init, MPI_Ssend_init И

MPI_Rsend_init. Список аргументов у них совпадает со списком аргументов подпрограммы MPI_Send_init.

Отложенный обмен инициализируется последующим вызовом подпрограммы MPI_Start:

int MPI_Start(MPI_Request ^request) MPI_START(REQUEST, IERR)

Входным параметром этой подпрограммы является запрос на выполнение операции обмена (request). Вызов MPi_start с запросом на обмен, созданным MPi_Send_init, инициирует обмен с теми же свойствами, что и вызов подпрограммы MPi_isend, а вызов MPi_start с запросом, созданным MPi_Bsend_init, инициирует обмен аналогично вызову MPi_ibsend. Сообщение, которое передано операцией, инициированной с помощью MPi_start, может быть принято любой подпрограммой приема.

Подпрограмма MPi_startall:

int MPI_Startall(int count, MPI_request *requests)

MPI_STARTALL(COUNT, REQUESTS, IERR)

инициирует все обмены, связанные с запросами на выполнение неблокирующей операции обмена в массиве requests. Завершается обмен при вызове MPI_wait, MPI_Test и некоторых других подпрограмм.

Отмена "ждущих" обменов

Операция MPi_Cancei позволяет аннулировать неблокирующие "ждущие" (т. е. ожидающие обработки) обмены. Ее можно использовать для освобождения ресурсов, прежде всего памяти, отведенной для размещения буферов:

int MPI_Cancel(MPI_request ^request)

MPI_CANCEL(REQUEST, IERR)

Вызов этой подпрограммы завершается сразу, возможно, еще до реального аннулирования обмена. Аннулируемый обмен следует завершить. Сделать это делается с помощью подпрограмм MPI_Request_free, MPI_Wait,

MPI_Test и некоторых других.

Операцию MPi_cancei можно использовать для аннулирования обменов, использующих как отложенный, так и обычный запрос. После вызова MPi_cancei и следующего за ним вызова MPi_wait или MPi_Test запрос на выполнение операции обмена становится неактивным и может быть активизирован для нового обмена. Информация об аннулированной операции содержится в аргументе status.

Подпрограмма MPI_Test_cancelled возвращает значение флага (flag) "истина", если

обмен, связанный с указанным статусом, успешно аннулирован:

int MPI_Test_cancelled(MPI_Status *status, int *flag)

MPI_TEST_CANCELLED(STATUS, FLAG, IERR)

Аннулирование — трудоемкая операция, не стоит ей злоупотреблять. Запрос на выполнение операции (request) можно аннулировать с помощью подпрограммы MPI_Request_f reel

int MPI_Request_free(MPI_Request *request)

MPI_REQUEST_FREE(REQUEST, IERR}

При вызове эта подпрограмма помечает запрос на обмен для удаления и присваивает ему значение MPI_REQUEST_NULL. Операции обмена, связанной с этим запросом, дается возможность завершиться, а сам запрос удаляется только после завершения обмена.

Следует учитывать, что после того как запрос аннулирован, нельзя проверить, был ли успешно завершен соответствующий обмен, поэтому активный запрос нельзя аннулировать.

Пример использования подпрограммы MPi_Request_free приведен в листинге 4.10.

Листинг 4.10. Пример использования подпрограммы MPi_Request_free

```
PROGRAM MAIN_MPI

INCLUDE 'mpif.h'

INTEGER RANK, TAG, CNT, IERR, N, STATUS(MPI_STATUS_SIZE)

REAL SNDBUF, RCVBUF

CNT = 1

TAG = 0

SNDBUF = 3.14159

RCVBUF = 0

CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
```

```
N = 4
IF(RANK.EQ.O) THEN
DO I=1, N
CALL MPI_ISEND(SNDBUF, CNT, MPI_REAL, 1, TAG,
MPI_COMM_WORLD, REQ, IERR)
PRINT *, "PROCESS ", RANK, " SEND ", SNDBUF
CALL MPI_REQUEST_FREE(REQ, IERR)
CALL MPI_IRECV(RCVBUF, CNT, MPI_REAL, 1, TAG,
MPI_COMM_WORLD, REQ, IERR)
PRINT *, "PROCESS ", RANK, " YET NOT RECEIVED ", RCVBUF
CALL MPI_WAIT(REQ, STATUS, IERR)
PRINT *, "PROCESS ", RANK, " RECEIVED ", RCVBUF
RCVBUF = 0
END DO
ELSE
CALL MPI_IRECV(RCVBUF, CNT, MPI_REAL, 0,
TAG, MPI_COMM_WORLD, REQ, IERR)
PRINT *, "PROCESS ", RANK, " YET NOT RECEIVED ", RCVBUF
CALL MPI_WAIT(REQ, STATUS, IERR)
PRINT *, "PROCESS ", RANK, " RECEIVED ", RCVBUF
RCVBUF = 0
DO I=1, N - 1
CALL MPI_ISEND(SNDBUF, CNT, MPI_REAL, 0,
TAG, MPI_COMM_WORLD, REQ, IERR)
```

```
PRINT *, "PROCESS ", RANK, " SEND ", SNDBUF
CALL MPI_REQUEST_FREE(REQ, IERR)
CALL MPI_IRECV(RCVBUF, CNT, MPI_REAL, 0,
TAG, MPI_COMM_WORLD, REQ, IERR)
PRINT *, "PROCESS ", RANK, " YET NOT RECEIVED ", RCVBUF
CALL MPI_WAIT(REQ, STATUS, IERR)
PRINT *, "PROCESS ", RANK, " RECEIVED ", RCVBUF
RCVBUF = 0
END DO
CALL MPI_ISEND(SNDBUF, CNT, MPI_REAL, 0, TAG,
MPI_COMM_WORLD, REQ, IERR)
PRINT *, "PROCESS ", RANK, " SEND ", SNDBUF
CALL MPI_WAIT(REQ, STATUS, IERR)
END IF
CALL MPI_FINALIZE(IERR)
STOP
END
```

Результат выполнения этой программы следующий:

```
# mpirun -np 2 a.out
Process 1 yet not received 0.
Process 1 received 3.14159012
Process 1 send 3.14159012
Process 1 yet not received 0.
Process 1 received 3.14159012
```

Process 1 send 3.14159012

Process 1 yet not received 0.

Process 1 received 3.14159012

Process 1 send 3.14159012

Process 1 yet not received 0.

Process 1 received 3.14159012

Process 1 send 3.14159012

Process 0 send 3.14159012

Process 0 yet not received 0.

Process 0 received 3.14159012

Process 0 send 3.14159012

Process 0 yet not received 0.

Process 0 received 3.14159012

Process 0 send 3.14159012

Process 0 yet not received 0.

Process 0 received 3.14159012

Process 0 send 3.14159012

Process 0 yet not received 0.

Process 0 received 3.14159012

Прокомментируйте эту распечатку.

Вызов подпрограммы MPI_Wait, который завершает прием, прерывается, если начинается прием сообщения. Если передача неблокирующая, прием будет завершен, даже если источник не вызывал подпрограмму завершения передачи. Аналогично, вызов MPI_Wait завершится, если начнется прием сообщения. Пример приведен в листинге 4.11.

Листинг 4.11. Использование подпрограммы MPI_Wait

```
PROGRAM MAIN_MPI
INCLUDE 'mpif.h'
INTEGER RANK, TAG1, TAG2, CNT, IERR, STATUS (MPI_STATUS_SIZE)
INTEGER REQUEST
REAL SNDBUF1, SNDBUF2, RCVBUF1, RCVBUF2
CNT = 1
TAG = 0
SNDBUF1 = 3.14159
SNDBUF2 = 2.71828
CALL MPI INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
IF (RANK.EQ.0) THEN
CALL MPI_SSEND(SNDBUF1, CNT, MPI_REAL, 1,
TAG1, MPI_COMM_WORLD, IERR)
PRINT *, "PROCESS ", RANK, " SEND ", SNDBUF1
CALL MPI_SEND(SNDBUF2, CNT, MPI_REAL, 1,
TAG2, MPI_COMM_WORLD, IERR)
PRINT *, "PROCESS ", RANK, " SEND ", SNDBUF2
ELSE
CALL MPI_IRECV(RCVBUF1, CNT,
MPI_REAL, 0, TAG1, MPI_COMM_WORLD, REQUEST,
IERR)
CALL MPI_RECV(RCVBUF2, CNT,
```

```
MPI_REAL, 0, TAG2, MPI_COMM_WORLD, STATUS,

IERR)

PRINT *, "PROCESS ", RANK, "

RECEIVED BEFORE WAIT", RCVBUF1

PRINT *, "PROCESS ", RANK, "

RECEIVED BEFORE WAIT", RCVBUF2

CALL MPI_WAIT(REQUEST, STATUS, IERR)

PRINT *, "PROCESS ", RANK, "

RECEIVED AFTER WAIT", RCVBUF1

PRINT *, "PROCESS ", RANK, "

RECEIVED AFTER WAIT", RCVBUF2

END IF

CALL MPI_FINALIZE(IERR)

STOP

END
```

При выполнении этой программы первая синхронная передача процессом с рангом 0 должна завершиться после того, как процесс с рангом 1 зарегистрирует соответствующий неблокирующий прием, даже если дело еще не дошло до вызова подпрограммы MPI_Wait. Процесс с рангом 0 выполнит вторую передачу, дав возможность процессу с рангом 1 завершить передачу. Вывод программы выглядит так:

```
# mpirun -np 2 a.out

Process 0 send 3.14159012

Process 0 send 2.71828008

Process 1 received before wait 2.08672428

Process 1 received before wait 2.71828008
```

Process 1 received after wait 3.14159012

Process 1 received after wait 2.71828008

Еще один пример использования неблокирующих операций и подпрограммы MPI_Wait приведен в листинге 4.12.

Листинг 4.12. Пример использования неблокирующих операций обмена

```
PROGRAM MAIN_MPI

INCLUDE 'mpif.h'

INTEGER RANK, TAG, CNT, IERR, STATUS(MPI_STATUS_SIZE)

INTEGER REQUEST

REAL SNDBUF(5) /1., 2., 3., 4., 5./

REAL RCVBUF(5)

CNT = 5

TAG = 0

CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)

IF(RANK.EQ.0) THEN

CALL MPI_ISEND(SNDBUF(1), CNT, MPI_REAL, 1, TAG,

MPI_COMM_WORLD, REQUEST,

IERR)

PRINT *, "PROCESS ", RANK, " SEND BEFORE WAIT", SNDBUF

CALL MPI_WAIT(REQUEST, STATUS, IERR}

PRINT *, "PROCESS ", RANK, " SEND AFTER WAIT", SNDBUF

ELSE

CALL MPI_IRECV(RCVBUF(1), CNT, MPI_REAL, 0, TAG,
```

```
MPI_COMM_WORLD, REQUEST,

IERR)

PRINT *, "PROCESS ", RANK, " RECEIVED BEFORE WAIT", RCVBUF

CALL MPI_WAIT(REQUEST, STATUS, IERR)

PRINT *, "PROCESS ", RANK, " RECEIVED AFTER WAIT", RCVBUF

END IF

CALL MPI_FINALIZE(IERR)

STOP

END
```

Результат выполнения этой программы:

```
# mpirun -np 2 a.out

Process 0 send before wait 1. 2. 3. 4. 5.

Process 0 send after wait 1. 2. 3. 4. 5.

Process 1 received before wait 5.73971851E-42 0. 4.12288481E-34

2.12725878 2.08672428

Process 1 received after wait 1. 2. 3. 4. 5.
```

Разберите работу программы и прокомментируйте результат ее выполнения.

Коллективный обмен данными

В двухточечном обмене участвуют два процесса — источник сообщения и его адресат. При выполнении коллективного обмена количество "действующих лиц" возрастает. Сообщение пересылается от одного процесса нескольким или, наоборот, один процесс "собирает" данные от нескольких процессов. В *главе 3* мы уже говорили о том, что MPI поддерживает такие виды коллективного обмена, как широковещательную передачу, операции приведения и т. д. В MPI имеются подпрограммы, выполняющие операции распределения и сбора данных, глобальные математические операции, такие
как суммирование элементов массива или вычисление его максимального элемента и т. д.

В любом коллективном обмене участвует каждый процесс из некоторой области взаимодействия. Можно организовать обмен и в подмножестве процессов, для этого имеются средства создания новых областей взаимодействия и соответствующих им коммуникаторов. Коллективные обмены характеризуются следующим:

- коллективные обмены не могут взаимодействовать с двухточечными. Коллективная передача, например, не может быть перехвачена двухточечной подпрограммой приема;
- коллективные обмены могут выполняться как с синхронизацией, так и без нее;
- все коллективные обмены являются блокирующими для инициировавшего их обмена;
- теги сообщений назначаются системой.

Широковещательная рассылка

Широковещательная рассылка выполняется одним выделенным процессом, который называется главным (root), а все остальные процессы, принимающие участие в обмене, получают по одной копии сообщения от главного процесса (рис. 4.6).

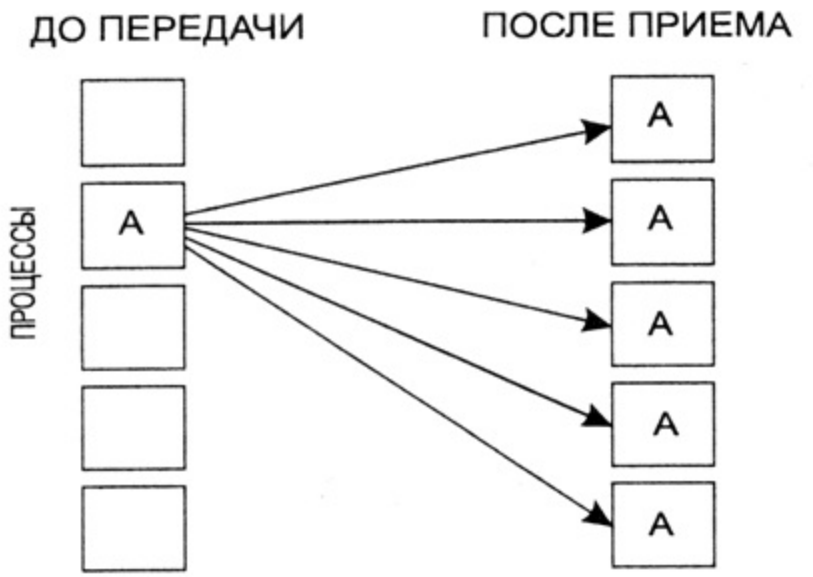


Рис. 4.6. *Широковещательная рассылка*

Выполняется широковещательная рассылка с помощью подпрограммы

MPI_Bcast:

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,

MPI_Comm comm)

MPI BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERR)

Ее параметры одновременно являются входными и выходными:

- buffer — адрес буфера;
- count — количество элементов данных в сообщении;
- datatype — тип данных MPI;
- root — ранг главного процесса, выполняющего широковещательную рассылку;
- comm — коммуникатор.

Обмен с синхронизацией

Синхронизация с помощью "барьера" является простейшей формой синхронизации коллективных обменов. Она не требует пересылки данных. Подпрограмма MPi_Barrier блокирует выполнение каждого процесса из коммуникатора comm до тех пор, пока все процессы не вызовут эту подпрограмму:

int MPI_Barrier(MPI_Comm comm} MPI_BARRIER(COMM, IERR)

Пример использования коллективного обмена приведен в листинге 4.13.

Листинг 4.13. Пример использования коллективного обмена

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

char data[24];

int myrank, count = 25;

MPI_Status status;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, Smyrank);

if (myrank == 0)

strcpy(data, "Hi, Parallel Programmer!");

MPI_Bcast (Sdata, count, MPI_BYTE, 0, MPI_CO^fXI_WORLD) ;

printf("send: %s\n", data);

else
```

MPI_Bcast(Sdata, count, MPI_BYTE, 0,

MPI_COMM_WORLD); printf("received: %s\n", data);

}

MPI_Finalize(); return 0; }

Результат выполнения этой программы:

mpirun -np 3 a.out send: Hi, Parallel Programmer! received: Hi,

Parallel Programmer! received: Hi, Parallel Programmer!

Управление областью взаимодействия и группой процессов

В MPI имеются средства создания и преобразования коммуникаторов, которые дают возможность программисту в дополнение к стандартным предопределенным объектам создавать свои собственные. Это не только позволяет использовать разнообразные и достаточно сложные схемы взаимодействия процессов, но и обеспечивает поддержку модульности в параллельных MPI-программах. Модульность — это возможность создавать библиотеки (библиотечные модули) параллельных подпрограмм. Читатель уже знает, что в таких подпрограммах используются аргументы, позволяющие определить источник и адресат обмена, а также теги сообщений. Теги позволяют различать сообщения, отправленные для разных целей, но механизма тегов недостаточно для поддержки модульности. Рассмотрим пример. Пусть программа использует библиотечную подпрограмму из созданного программистом модуля параллельных процедур. Для правильной организации обмена сообщениями важно, чтобы теги сообщений, применяемые в библиотеке, отличались от тегов, используемых в оставшейся части приложения. Но пользователь библиотечной процедуры может и не знать, какие теги она использует. В этом случае возникает опасность создания нестабильно работающей параллельной профаммы с непредсказуемым поведением.

Решением этой проблемы и являются коммуникаторы. Используя средства MPI, можно создать новый коммуникатор, содержащий, например, те же процессы, что и исходный, но с новым контекстом (т. е. с новыми свойствами). Обмен возможен только в рамках одного контекста, а обмены в разных коммуникаторах происходят независимо. Новый коммуникатор можно передать в качестве аргумента библиотечной подпрофамме, как в следующем примере:

CALL MPI_COMM_DUP(COMM, NEWCOMM, IERR)

CALL GAUSS_PARALLEL(NEWCOMM, A, B)

CALL MPI COMM FREE(NEWCOMM, IERR)

Здесь сначала создается новый коммуникатор (о подпрограмме MPI_Comm_dup речь пойдет ниже), подпрограмма gauss_parallel использует коммуникатор newcomm так, что все обмены в ней гарантированно выполняются независимо от других операций обмена.

Детали внутреннего представления коммуникаторов и групп зависят от конкретной реализации, пользователь не имеет к ним прямого доступа. В MPI существует механизм "кэширования", который позволяет связать с коммуникатором новый набор атрибутов. Это "высший пилотаж" программирования на **MPI.**

Группы процессов

Группой называют упорядоченное множество процессов. Каждому процессу в группе сопоставлен свой ранг. Операции с группами могут выполняться отдельно от операций с коммуникаторами, но в операциях обмена используются только коммуникаторы. В MPI имеется специальная предопределенная пустая группа MPI_GROUP_EMPTY.

Коммуникаторы бывают двух типов: *интракоммуникаторы* — для операций внутри одной группы процессов и *интеркоммуникаторы* — для двухточечного обмена между двумя группами процессов. Интракоммуникатором, в частности, является MPI_COMM_WORLD.

В MPI-программах чаще используются интракоммуникаторы. Интракомму-никатор включает экземпляр группы, контекст обмена для всех его видов, а также, возможно, виртуальную топологию (о топологиях мы поговорим в следующей главе) и другие атрибуты. Контекст обеспечивает возможность создания изолированных друг от друга, а потому "безопасных", областей взаимодействия. Система сама управляет их разделением. Контекст играет роль дополнительного тега, который дифференцирует сообщения.

MPI поддерживает обмены между двумя непересекающимися группами процессов. Если параллельная программа состоит из нескольких параллельных модулей, удобно разрешить одному модулю обмениваться сообщениями с другим, используя для адресации локальные по отношению ко второму модулю ранги. Такой подход удобен, например, при программировании параллельных клиент-серверных приложений. Интеробмены реализуются с помощью интеркоммуникаторов, которые объединяют две группы процессов общим контекстом. В контексте, связанном с интеркоммуникатором, передача сообщения в локальной группе всегда сопровождается его приемом в удаленной группе — это двухточечная операция. Группа, содержащая процесс, который инициирует операцию интеробмена, называется *локальной группой,* а группа, содержащая процесс-адресат, — *удаленной группой.* Интеробмен задается парой: коммуникатор — ранг, при этом ранг задается

относительно удаленной группы.

Конструкторы интеркоммуникаторов (конструктор - это подпрограмма, создающая соответствующие структуры данных) являются блокирующими операциями, поэтому во избежание тупиковых ситуаций локальная и удаленная группа не должны пересекаться, т. е. они не должны содержать одинаковые процессы.

Интеробмен характеризуется следующими свойствами:

- синтаксис двухточечных обменов одинаков в операциях обмена в рамках интра- и интеркоммуникаторов;
- адресат задается рангом процесса -в удаленной группе;
- операции обмена с использованием интеркоммуникаторов не вступают в конфликт с обменами, использующими другой коммуникатор;
- интеркоммуникатор нельзя использовать для коллективных обменов;
- коммуникатор не может объединять свойства интер- и интракоммуника-тора.

Интеркоммуникатор создается коллективным вызовом подпрограммы MPI_Intercomm_create (подробнее об этой подпрограмме поговорим позже).

Создание групп процессов

Операции создания групп процессов аналогичны математическим операциям над множествами:

- *объединение* — к процессам первой группы (group1) добавляются процессы второй группы (group2), не принадлежащие первой;
- *пересечение* — в новую группу включаются все процессы, принадлежащие двум группам одновременно. Ранги им назначаются как в первой группе;
- *разность* — в новую группу включаются все процессы первой группы, не входящие во вторую группу. Ранги назначаются как в первой группе.

Новую группу можно создать только из уже существующих групп. Базовая группа, из которой формируются все другие группы, связана с коммуникатором MPI_COMM_WORLD. В подпрограммах создания групп, как правило, нельзя использовать пустой коммуникатор MPI_COMM_NULL.

Доступ к группе group, связанной с коммуникатором comm, можно получить, обратившись к подпрограмме MPi_comm_group:

int MPI_Coram_group(MPI_Comm comn, MPI_Group *group)

MPI_COMM_GROUP(COMM, GROUP, IERR)

Ее выходным параметром является группа. Для выполнения операций с группой к ней сначала необходимо получить доступ.

Подпрограмма MPi_Group_inci создает новую группу newgroup из п процессов, входящих в группу oidgroup. Ранги этих процессов содержатся в массиве ranks:

int MPI_Group_incl(MPI_Group oidgroup, int n, int *ranks,

MPI_Group *newgroup)

MPI_GROUP_INCL(OLDGROUP, N, RANKS, NEWGROUP, IERR)

В новую группу войдут процессы с рангами ranks [0], ..., ranks [n - 1], причем рангу i в новой группе соответствует ранг ranks [i] в старой группе. При п = о создается пустая группа MPI_GROUP_EMPTY. С помощью данной подпрограммы можно не только создать новую группу, но и изменить порядок процессов в старой группе.

Подпрограмма MPI_Group_excl создает группу newgroup, исключая из исходной группы (group) процессы с рангами ranks [0], ..., ranks [n - 1]:

int MPI_Group_excl(MPI_Group oidgroup, int n, int *ranks,

MPI_Group *newgroup)

MPI_GROUP_EXCL(OLDGROUP, N, RANKS, NEWGROUP, IERR)

При n = о новая группа тождественна старой.

Подпрограмма MPI_Group_range_ind создает группу newgroup из группы group, добавляя в нее п процессов, ранг которых указан в массиве ranks:

int MPI_Group_range_incl(MPI_Group oidgroup, int n, int ranks[][3],

MPI_Group *newgroup)

MPI_GROUP_RANGE_INCL(OLDGROUP, N, RANKS, NEWGROUP, IERR)

Массив ranks состоит из целочисленных триплетов вида (первый_1, последний_1, шаг_1), ..., (первый_n, последний_п, шаг_п). В новую группу войдут процессы с рангами (по первой группе) первый_1, первый_1 + шаг_1, ...

Подпрограмма MPi_Group_range_exci создает группу newgroup из группы group, исключая из нее п процессов, ранг которых указан в массиве ranks:

int MPI_Group_range_excl(MPI_Group group, int n, int ranks[][3],

MPI_Group *newgroup)

MPI_GROUP_RANGE_EXCL(GROUP, N, RANKS, NEWGROUP, IERR)

Массив ranks устроен так же, как аналогичный массив в подпрограмме

MPI_Group_range_incl.

Подпрограмма MPI_Group_difference создает новую группу (newgroup) из разности двух групп (groupl) и (group2):

int MPI_Group_difference(MPI_Group groupl, MPI_Group group2, MPI_Group

*newgroup)

MPI GROUP DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERR)

Подпрограмма MPI_Group_intersection создает новую фуппу (newgroup) из пересечения групп group1 и group2:

int MPI_Group_intersection(MPI_Group groupi, MPI_Group group2,

MPI_Group *newgroup)

MPI_GROUP_INTERSECTION(GROUPI, GROUP2, NEWGROUP, IERR)

Подпрограмма MPI_Group_union создает группу (newgroup), объединяя группы

groupi И group2:

int MPI_Group_union(MPI_Group group1, ,MPI_Group group2, MPI_Group *newgroup)

MPI_GROUP_UNION(GROUPI, GROUP2, NEWGROUP, IERR)

Имеются и другие подпрограммы-конструкторы новых jpynn. Есть и деструктор. Вызов подпрограммы MPI_Group_free уничтожает группу group:

int MPI_Group_f ree (MPI__Group *group) MPI GROUP FREE(GROUP, IERR)

Получение информации о группе

Для определения количества процессов (size) в группе (group) используется подпрограмма MPI_Group_size:

int MPI_Group_size(MPI_Group group, int *size) MPI_GROUP_SIZE(GROUP, SIZE, IERR)

Подпрограмма MPi_Group_rank возвращает ранг (rank) процесса в группе group:

int MPI_Group_rank(MPI_Group group, int *rank) MPI_GROUP_RANK(GROUP, RANK, IERR)

Если процесс не входит в указанную группу, возвращается значение

MPI_UNDEFINED.

Процесс может входить в несколько групп. Подпрограмма MPI_Group_ transiate_ranks преобразует ранг процесса в одной группе в его ранг в другой группе:

int MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1,

MPI_Group group2, int *ranks2)

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERR)

Эта функция используется для определения относительной нумерации одних и тех же процессов в двух разных группах.

Подпрограмма MPi_Group_compare используется для сравнения групп groupi

И group2:

int MPI_Group_compare(MPI_Group groupi, MPI_Group group2, int *result) MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERR)

Если группы полностью совпадают, возвращается значение MPI_IDENT. Если члены обеих групп одинаковы, но их ранги отличаются, результатом будет значение MPI_SIMILAR. Если группы различны, результатом будет MPI_UNEQUAL.

Управление коммуникаторами

Создание коммуникатора - коллективная операция и соответствующая подпрограмма должна вызываться всеми процессами коммуникатора. Подпрограмма Mpi_comm_dup дублирует уже существующий коммуникатор oldcomm:

int MPI_Comm_dup(MPI_Comm oldcomm, MPI_Comm *newcomm)

MPI_COMM_DUP (OLDCOMM, NEWCOMM, IERR)

В результате вызова создается новый коммуникатор (newcomm) с той же группой процессов, с теми же атрибутами, но с другим контекстом. Подпрограмма может применяться как к интра-, так и к интеркоммуникаторам.

Подпрограмма MPI_Comm__create создает новый коммуникатор (newcomm) из подмножества процессов (group) другого коммуникатора (oldcomm):

int MPI_Comm_create(MPI_Comm oldcomm, MPI_Group group, MPI_Comm *newcomm)

MPI_COMM_CREATE(OLDCOMM, GROUP, NEWCOMM, IERR)

Вызов этой подпрограммы должны выполнить все процессы из старого коммуникатора, даже если они не входят в группу group, с одинаковыми аргументами. Данная операция применяется только к интракоммуникато-рам. Она позволяет выделять подмножества процессов со своими областями взаимодействия, если, например, требуется уменьшить "зернистость" параллельной программы. Побочным эффектом применения подпрограммы MPi_comm_create является синхронизация процессов. Если, например, одновременно создаются несколько коммуникаторов, они должны создаваться в одной последовательности всеми процессами. Пример создания коммуникатора приведен в листинге 4.14.

Листинг 4.14. Пример создания коммуникатора

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

char message[24];

MPI_Group MPI_GROUP_WORLD;

MPI_Group group;

MPI_Comm fcomm;

int size,' q, proc;

int* process_ranks;

int rank, rank_in_group;

MPI_Status status; MPI_Init(&argc, Sargv);

MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

printfC'New group contains processes:");
```

```
q = size — 1;

process_ranks = (int*) malloc(q*sizeof(int)); for (proc = 0; proc < q; proc++)

{

process_ranks[proc] = proc; printf("%i ", process_ranks[proc]); >

printf("\n");

MPI_Coinm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks, Sgroup);
MPI_Comm_create(MPI_COMM_WORLD, group, Sfcoram);

if (fcomm != MPI_COMM_NULL) { MPI_Comm_group(group, &fcomm);

MPI_Comm_rank(fcomm, & rank_in_group);

if (rank_in_group ==0) {

strcpy(message, "Hi, Parallel Programmer!");

MPI_Bcast(Smessage, 25, MPI_BYTE, 0, fcomm);

printf("0 send: %s\n", message); } else

{

MPI_Bcast(Smessage, 25, MPI_BYTE, 0, fcomm);

printf("%i received: %s\n", rank_in_group, message);

}

MPI_Comm_free(&fcomm); MPI_Group_free(&group);

}

MPI_Finalize(); return 0;
```

Посмотрим, как работает эта программа. Пусть в коммуникатор MPI_ COMM_WORLD входят р процессов. Сначала создается список процессов, которые будут входить в область взаимодействия нового коммуникатора. Затем создается группа, состоящая из этих процессов. Для этого требуются две операции. Первая определяет группу, связанную с коммуникатором MPI_ COMM_WORLD. Новая группа создается вызовом

подпрограммы MPI_Group_incl. Затем создается новый коммуникатор. Для этого используется подпрограмма MPI_Comm_create. Новый коммуникатор — fcomm. В результате всех этих действий все процессы, входящие в коммуникатор fcomm, смогут выполнять операции коллективного обмена, но только между собой. Результат ее выполнения:

mpirun -np 5 a.out

New group contains processes:0 123

0 send: Hi, Parallel Programmer!

New group contains processes:0 1 2 3

1 received: Hi, Parallel Programmer!

New group contains processes:0 1 2 3 3 received: Hi,

Parallel Programmer! New group contains processes:0 123

2 received: Hi, Parallel Programmer!

Подпрограмма MPI_comm_spiit позволяет создать несколько коммуникаторов сразу:

int MPI_Comm_split(MPI_Comm oldcomm, int split, int rank, MPI_Comm* newcomm)

MPI_COMM_SPLIT(OLDCOMM, SPLIT, RANK, NEWCOMM, IERR)

Группа процессов, связанных с коммуникатором oldcomm, разбивается на непересекающиеся подгруппы, по одной для каждого значения аргумента split. Процессы с одинаковым значением split образуют новую группу. Ранг в новой группе определяется значением rank. Если процессы А и В вызывают MPI_comm_split с одинаковым значением split, а аргумент rank, переданный процессом А, меньше, чем аргумент, переданный процессом В, ранг А в группе, соответствующей новому коммуникатору, будет меньше ранга процесса В. Если же в вызовах используется одинаковое значение rank, система присвоит ранги произвольно. Для каждой подгруппы создается Собственный КОММунИКатОр newcomm.

MPI_Comm_split — коллективная подпрограмма, ее должны вызвать все процессы из старого коммуникатора, даже если они и не войдут в новый коммуникатор. Для этого в качестве аргумента split в подпрограмму передается предопределенная константа MPI_UNDEFINED. Соответствующие процессы вернут в качестве нового коммуникатора значение MPI_COMM_NULL. Новые коммуникаторы, созданные подпрограммой MPI_comm_split, не пересекаются, подпрограммы MPI_Comm_split можно создавать и перекрывающиеся коммуникаторы. Пример расщепления

коммуникатора:

CALL MPI_COMM_RANK(COMM, RANK, IERR)

CALL MPI_COMM_SIZE(COMM, SIZE, IERR)

SPLIT = 2 * RANK / SIZE

KEY = SIZE - RANK - 1

CALL MPI_COMM_SPLIT(COMM, SPLIT, KEY, NEWCOMM, IERR)

В следующем примере создаются три новых коммуникатора (если исходный коммуникатор comm содержит не менее трех процессов):

MPI_Comm comm, newcomm;

int rank, split;

MPI_Comm_rank(comm, srank);

split = rank%3;

MPI_Comm_split(comm, split, rank, Snewcomm) ;

Если, например, коммуникатор comm содержит 7 процессов, будет создан новый коммуникатор из процессов 0, 2 и 3; коммуникатор из процессов 1, 4 и 6, а также коммуникатор из одного процесса 5 (рис. 4.7).

Подпрограмма MPi_Comm_f гее помечает коммуникатор comm для удаления:

int MPI_Comm_free(MPI_Comm *comm) MPI COMM FREE(COMM, IERR)

Рис. 4.7. *Создание трех новых коммуникаторов с помощью подпрограммы MPI_Comm_split*

Обмены, связанные с этим коммуникатором, завершаются обычным образом, а сам коммуникатор удаляется только после того, как на него не будет активных ссылок. Данная операция может применяться к коммуникаторам обоих видов (интра- и интер-).

Сравнение двух коммуникаторов (commi) и (comm2) выполняется подпрограммой MPI_Comm_compare:

int MPI_Comm_compare(MPI_Comm comml,

MPI_Comm comm2, int *result)

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERR)

Результат ее выполнения result — целое значение, которое равно MPI_IDENT, если контексты и группы коммуникаторов совпадают; MPI_CONGRUENT, если совпадают только группы; MPI_SIMILAR и MPI_UNEQUAL, если не совпадают ни группы, ни контексты. В качестве аргументов нельзя использовать пустой коммуникатор MPI_COMM_NULL.

К числу операций управления коммуникаторами можно отнести знакомые уже нам операции MPI_Comm_Size и MPI_Comm_rank. Они позволяют, в частности, распределить роли между процессами в модели "главная задача — подчиненные

задачи" (master-slave).

С помощью подпрограммы MPi_Comm_set_name можно присвоить коммуника-тору comm строковое имя name:

int MPI_Comm_set_name(MPI_Comm com, char *name) MPI_COMM_SET_NAME(COM, NAME, IERR)

и наоборот, подпрограмма MPi_Comm_get_name возвращает name — строковое имя коммуникатора comm:

int MPI_Comm_get_name(MPI_Comm comm, char *name, int *reslen) MPI_COMM_GET_NAME(COMM, NAME, RESLEN, IERR)

Имя представляет собой массив символьных значений, длина которого должна быть не- менее MPI_MAX_NAME_STRING. Длина имени — выходной параметр reslen.

Проверка, является ли коммуникатор comm (это входной параметр) интеркоммуникатором, выполняется подпрограммой MPi_Comm_test_inter:

int MPI_Comm_test_inter(MPI_Comm comm, int *flag) MPI_COMM_TEST_INTER(COMM, FLAG, IERR)

Результатом является значение флага (flag) "истина", если аргументом является интеркоммуникатор.

Две области взаимодействия можно объединить в одну. Подпрограмма MPi_intercomm_merge создает интракоммуникатор newcomm из интеркоммуни-катора oldcomm:

int MPI_Intercomm_merge(MPI_Comm oldcomm, int high, MPI_Comm *newcomm) MPI_INTERCOm_MERGE(OLDCOMM, HIGH, NEWCOMM, IERR)

Параметр high здесь используется для упорядочения групп обоих интракоммуникаторов в comm при создании нового коммуникатора.

Доступ к удаленной группе, связанной с интеркоммуникатором comm, можно получить, обратившись к подпрограмме MPI_comm_remote_group:

int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group) MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERR)

Ее выходным параметром является удаленная группа group.

Подпрограмма MPi_comm_remote_size определяет размер удаленной группы, связанной с интеркоммуникатором comm:

int MPI_Comm_remote_size(MPI_Comm comm, int *size)
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERR)

Ее выходной параметр size — количество процессов в области взаимодействия, связанной с коммуникатором comm.

Интеркоммуникаторы можно использовать в операциях, которые обычно применяются к интракоммуникаторам. При этом MPI_Comm_size возвращает размер локальной группы, MPi_Comm_group - локальную группу и MPi_comm_rank возвращает ранг в локальной группе. В подпрограмме MPi_comm_compare оба коммуникатора должны быть одного типа, иначе результатом будет значение MPIJJNEQUAL.

Операции обмена между группами процессов (интеробмен)

При выполнении интеробмена процессу-источнику сообщения указывается ранг адресата относительно удаленной группы, а процессу-получателю -ранг источника (также относительно удаленной по отношению к получателю группы). Обмен выполняется между лидерами обеих групп. Предполагается, что в обеих группах есть, по крайней мере, по одному процессу, который может обмениваться сообщениями со своим партнером.

Интеробмен возможен, только если создан соответствующий интеркоммуникатор, а это можно сделать с помощью подпрограммы MPi_intercomm_create!

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,

MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *new_intercomm)

MPI_INTERCOMM_CREATE(LOCAL_COMM,

LOCAL_LEADER, PEER_COMM, REMOTE_

LEADER, TAG, NEW INTERCOM, IERR)

Входные параметры этой подпрограммы:

- local_comm — локальный интракоммуникатор;
- local_leader — ранг лидера в локальном коммуникаторе (обычно 0);
- peer_comm — удаленный коммуникатор;
- remote_leader — ранг лидера в удаленном коммуникаторе (обычно 0);
- tag — тег интеркоммуникатора, используемый лидерами обеих групп для обменов в контексте родительского коммуникатора.

Выходной параметр — интеркоммуникатор (new_intercomm). "Джокеры" в качестве

параметров использовать нельзя. Вызов этой подпрограммы должен выполняться в обеих группах процессов, которые должны быть связаны между собой. При этом в каждом из этих вызовов используется локальный интракоммуникатор, соответствующий данной группе процессов. При работе с MPi_intercomm_create локальная и удаленная группы процессов не должны пересекаться, иначе возможны "тупики".

Пример организации обмена между разными группами с помощью интеркоммуникатора приведен в листинге 4.15.

Листинг 4.15. Пример создания интеркоммуникаторов

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

{

int counter, message, myid, numprocs, server;

int color, remote_leader_rank, i, ICTAG = 0;

MPI_Status status;

MPI_Comm oldcommdup, splitcomm, oldcomm, inter_comm;

MPI_Init(&argc, &argv);

oldcomm = MPI_COMM_WORLD;

MPI_Comm_dup(oldcomm, &oldcommdup);

MPI_Comm_size(oldcommdup, &numprocs);

MPI_Comm_rank(oldcommdup, &myid);

server = numprocs — 1;

color = (myid == server);

MPI_Comm_split(oldcomm, color, myid, Ssplitcomm);

if(!color) {
```

```
remote_leader_rank = server;

}

else { remoteleader_rank = 0;

MPI_Intercomm_create (splitcomm, 0, oldcommdup,

remote_leader_rank, ICTAG, &inter_comm) ;

MPI_Comm_free (soldcommdup) ; if (myid == server) { for(i = 0; i<server; i++) {

MPI_Recv(&message, 1, MPI_INT, i, MPI_ANY_TAG, inter_comm, Sstatus) ;

printf ("Process rank %i received %i from %i\n", myid, message, status. MPI_SOURCE) ; }

} else{

counter = myid;

MPI_Send(&counter, 1, MPI_INT, 0, 0, inter_comm) ;

printf ("Process rank %i send %i\n", counter); }

MPI_Comm_free (&inter_comm) ;

MPI_Finalize() ;

return 0 ;
```

В этой программе процессы делятся на две группы: первая состоит из одного процесса (процесс с максимальным рангом в исходном коммуникаторе MPI_COMM_WORLD), это — "сервер", а во вторую входят все остальные процессы. Между этими группами и создается интеркоммуникатор inter_comm. Процессы-клиенты передают серверу сообщения. Сервер принимает эти сообщения с помощью подпрограммы стандартного блокирующего двухточечного приема и выводит их на экран. "Ненужные" коммуникаторы удаляются. Распечатка вывода этой программы выглядит следующим образом:

mpirun -np 10 a. out Process rank 4 send 4 Process rank 8 send 8 Process rank 2 send 2 Process rank 6 send 6 Process rank 3 send 3 Process rank 1 send 1 Process rank 9 received 0 from 0 Process rank 9 received 1 from 1 Process rank 9 received 2 from 2 Process rank 9 received 3 from 3 Process rank 9 received 4 from 4 Process rank 9 received 5 from 5 Process rank 9 received 6 from 6 Process rank 9 received 7 from 7 Process rank 9 received 8 from 8

Process rank 5 send 5 Process rank 7 send 7 Process rank 0 send 0

Разберите работу программы и прокомментируйте результат ее выполнения.

Вопросы и задания для самостоятельной работы

1. Напишите программу, которая выполняется в трех процессах, и результат зависит от порядка приема сообщений.

2. Напишите программу, в которой два процесса многократно обмениваются сообщениями длиной *n* элементов. Проведите исследование зависимости времени выполнения программы от длины сообщения.

3. Запрограммируйте двумерный конечно-разностный алгоритм решения дифференциального уравнения и проведите измерения производительности программы для разного числа процессоров.

4. Напишите параллельную программу, которая выполняет транспонирование матрицы *NxN* на *M* процессорах. Каждому процессу передаются *N/M* строк, а после транспонирования каждый процесс возвращает *N/M* столбцов. Проведите исследование зависимости быстродействия этой программы от числа процессоров. Попробуйте использовать разные виды обмена и сравните результаты.

5. Напишите параллельную программу решения любого дифференциального уравнения сеточным методом, используя разные шаблоны.

6. Напишите параллельную программу вычисления максимального (минимального) элемента массива вещественных значений.

7. Напишите параллельную программу, которая выполняет перемножение двух матриц.

8. Напишите параллельную программу, в которой создаются *N* групп процессов, и обмен между этими группами выполняется по замкнутому кольцу.

ГЛАВА 5.

Коллективный обмен данными в MPI

В этой главе мы познакомимся с разновидностями коллективного обмена в MPI: распределением и сбором данных, глобальными операциями, операциями сканирования. Затем научимся конструировать производные типы и выполнять обмен сложными наборами данных. Далее в главе рассматриваются способы создания виртуальных топологий и назначение коммуникаторам атрибутов. В заключение дается обзор подпрограмм параллельного ввода/вывода из библиотеки ROMIO, и предлагаются задания для самостоятельной работы.

Разновидности коллективного обмена

Участниками коллективных обменов сообщениями являются все процессы из заданной области взаимодействия. Обеспечить распределение и сбор данных можно было бы и с помощью многократных двухточечных обменов, однако при этом пришлось бы применять достаточно сложные схемы пересылок данных. Организация обменов в рамках таких схем не очень проста и может требовать выполнения вспомогательных расчетов и операций, что значительно усложняет программирование и увеличивает его трудоемкость. Разработчику параллельной программы требуется средство организации обменов на множестве процессов, которое скрывало бы от него несущественные технические детали. Таким средством и являются коллективные обмены. Использование коллективных обменов позволяет, кроме того, избежать простоев процессов, которые были бы неизбежны при организации многократных пересылок данных с помощью двухточечных обменов.

В MPI определены следующие типы коллективных обменов сообщениями:

- широковещательная рассылка;
- сбор данных;
- распределение данных;
- операции приведения (или редукции) и сканирования.

Широковещательная пересылка

С этой разновидностью коллективного обмена мы уже познакомились в *главе 4.* Напомним, что при широковещательной передаче один и тот же набор данных передается каждому процессу из соответствующего коммуникатора. Для выполнения широковещательного обмена используется подпрограмма MPI_Bcast. Заметим, что при вызове этой подпрограммы в разных процессах должны использоваться одинаковые значения параметров count и datatype. Одна и та же подпрограмма служит для передачи и для приема данных.

Пример использования широковещательной пересылки для передачи трех числовых значений, вводимых с клавиатуры в одном процессе, всем остальным процессам из коммуникатора MPI_COMM_WORLD, приведен в листинге 5.1.

Листинг 5.1. Пример использования широковещательной пересылки

```
#include "mpi.h"

#include <stdio.h>

int main (int argc, char *argv[])

{

int my rank;

int root = 0;

int count = 1;

float a, b;

int n;

MPI_Init (&argc, &7argv) ;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank) ;

if (my rank == 0) {
```

```
printf ("Enter a, b, n\n"); scanf("%f %f %i", &a, &b, &n) ;

MPI_Bcast (&a, count, MPI_FLOAT, root,

MPI_COMM_WORLD) MPI_Bcast (&b, count,

MPI_FLOAT, root, MPI_COMM_WORLD)

MPI_Bcast(Sn, count, MPI_INT, root, MPI_COMM_WORLD) ;

}

else

}

MPI_Bcast(&a, count, MPI_FLOAT, root,

MPI_COMM_WORLD) ;

MPI Beast (Sb, count, MPI FLOAT, root, MPI COMM_WORLD) ;

MPI_Bcast (&n, count, MPI_INT, root, MPI_COMM_WORLD) ;

printf("%i Process got %f %f %i\n", myrank, a, b, n) }

MPI_Finalize() ;

return 0;
```

Распределение и сбор данных

Распределение и сбор данных выполняются с помощью подпрограмм MPI_Scatter и MPI_Gather соответственно. Список аргументов у обеих подпрограмм одинаков, но действуют они по-разному.

На рис. 5.1 и 5.2 приведены схемы передачи данных для операций сбора и распределения данных.

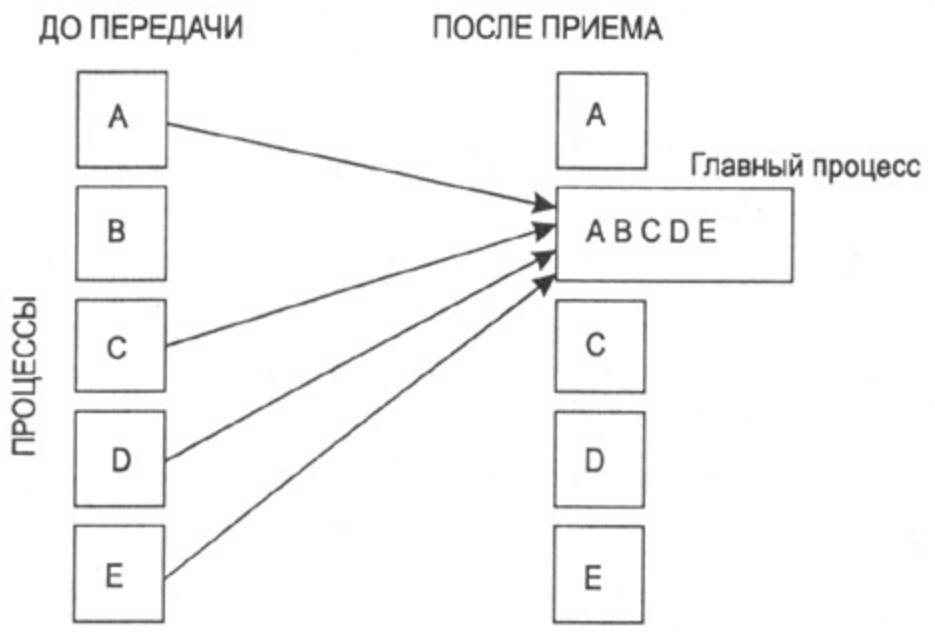


Рис. 5.1. *Схема передачи данных для операции сбора данных*

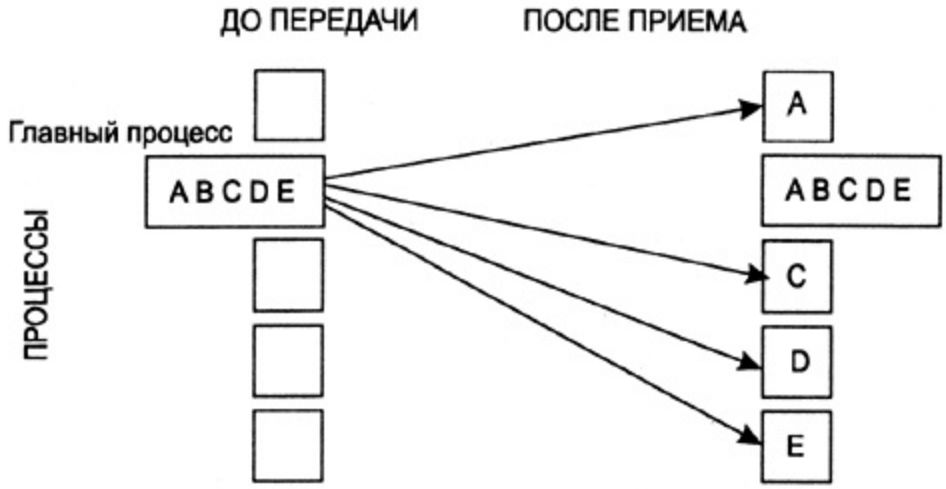


Рис. 5.2. *Схема передачи данных для операции распределения данных*

В табл. 5.1 приведен перечень подпрограмм распределения и сбора данных. ***Таблица 5.1.*** *Подпрограммы распределения и сбора данных*

Подпрограмма	**Краткое описание**
MPI_Allgather	Собирает данные от всех процессов и пересылает их всем процессам
	Собирает данные от всех процессов и пересылает их всем процессам

MPI Allgatherv	("векторный" вариант подпрограммы MPI_Allgather)
MPI Allreduce	Собирает данные от всех процессов, выполняет операцию приведения, и результат распределяет всем процессам
MPI_Alltoall	Пересылает данные от всех процессов всем процессам
MPI Alltoallv	Пересылает данные от всех процессов всем процессам ("векторный" вариант подпрограммы MPI Alltoall)
MPI_Gather	Собирает данные от группы процессов
MPI Gatherv	Собирает данные от группы процессов ("векторный" вариант подпрограммы MPI Gather)
MPI Reduce	Выполняет операцию приведения, т. е. вычисление единственного значения по массиву исходных данных
MPI Reduce scatter	Сбор значений с последующим распределением результата операции приведения
MPI_Scan	Выполнение операции сканирования (частичная редукция) для данных от группы процессов

MPI Scatter	Распределяет данные от одного процесса всем остальным процессам в группе
MPI Scatterv	Пересылает буфер по частям всем процессам в группе ("векторный" вариант подпрограммы MPI Scatter)

При широковещательной рассылке всем процессам передается один и тот же набор данных, а при распределении передаются его части. Выполняется распределение данных подпрограммой MPi_Scatter, которая пересылает данные от одного процесса всем остальным процессам в группе так, как это показано на рис. 5.2:

int MPI_Scatter(void *sendbuf, int sendcount,

MPI_Datatype sendtype, void *rcvbuf, int rcvcount,

MPI_Datatype rcvtype, int root, MPI_Comm coram)

MPI_SCATTER(SENDBUF, SENDCOUNT,

SENDTYPE, RCVBUF, RCVCOUNT,

RCVTYPE, ROOT, COMM, IERR)

Ее входные параметры:

- sendbuf — адрес буфера передачи;
- sendcount — количество элементов, пересылаемых каждому процессу (но не суммарное количество пересылаемых элементов);
- sendtype — тип передаваемых данных;
- rcvcount — количество элементов в буфере приема;
- rcvtype — тип принимаемых данных;
- root — ранг передающего процесса;
- comm — коммуникатор.

Выходной параметр rcvbuf — адрес буфера приема. Работает эта подпрограмма следующим образом. Процесс с рангом root ("главный процесс") распределяет содержимое буфера передачи sendbuf среди всех процессов. Содержимое буфера передачи разбивается на несколько фрагментов, каждый из которых содержит sendcount элементов. Первый фрагмент передается процессу 0, второй процессу 1 и т. д. Аргументы send имеют значение только на стороне процесса root.

Подпрограмма MPI_Gather выполняет сбор сообщений от остальных процессов в буфер главной задачи так, как это показано на рис. 5.1:

int MPI_Gather(void *sendbuf, int sendcount,

MPI_Datatype sendtype, void *rcvbuf, int rcvcount,

MPI_Datatype rcvtype, int root,

MPI_Comm comm) MPI_GATHER(SENDBUF,

SENDCOUNT, SENDTYPE, RCVBUF,

RCVCOUNT, RCVTYPE, ROOT, COMM, IERR)

Каждый процесс в коммуникаторе comm пересылает содержимое буфера передачи sendbuf процессу с рангом root. Процесс root "склеивает" полученные данные в буфере приема. Порядок склейки определяется рангами процессов, т. е. в результирующем наборе после данных от процесса 0 следуют данные от процесса 1, затем данные от процесса 2 и т. д. Аргументы rcvbuf, rcvcount и rcvtype играют роль только на стороне главного процесса. Аргумент rcvcount указывает количество элементов данных, полученных от каждого процесса (но не суммарное их количество). При вызове подпрограмм MPI_Scatter и MPI_Gather из разных процессов следует использовать общий главный процесс.

Подпрограммы MPI_Scatterv и MPi_Gatnerv являются расширенными ("векторными") версиями подпрограмм MPi_scatter и MPi_Gather. Они позволяют пересылать разным процессам (или собирать от них) разное количество элементов данных. Благодаря аргументу displs *(см. ниже),* который задает расстояние между элементами, пересылаемые элементы могут не располагаться непрерывно в памяти главного процесса. Это может оказаться полезным, например, при пересылке частей массивов.

Векторная подпрограмма распределения — MPI_scatterv:

int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,

MPI_Datatype sendtype, void *rcvbuf, int rcvcount,

MPI_Datatype rcvtype, int root, MPI_Comm comm)

MPI_SCATTERV(SENDBUF, SENDCOUNTS,

DISPLS, SENDTYPE, RCVBUF, RCVCOUNT,

RCVTYPE, ROOT, COMM, IERR)

Входные параметры:

- sendbuf — адрес буфера передачи;
- sendcounts -- целочисленный одномерный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата). Его длина равна количеству процессов в коммуникаторе;
- displs — целочисленный массив, длина которого равна количеству процессов в коммуникаторе. Элемент с индексом i задает смещение относительно начала буфера передачи. Ранг адресата равен значению индекса i;
- sendtype — тип данных в буфере передачи;
- rcvcount — количество элементов в буфере приема;
- rcvtype — тип данных в буфере приема;
- root — ранг передающего процесса;
- comm — коммуникатор.

Выходным является адрес буфера приема rcvbuf.

Подпрограмма Mpi_Gatherv используется для сбора данных от всех процессов в заданном коммуникаторе и записи их в буфер приема с указанным смещением:

int MPI_Gatherv(void *sendbuf, int sendcount,

MPI_Datatype sendtype, void *recvbuf, int *recvcounts,

int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE,

RECVBUF, RECVCOUNTS, DISPLS,

RECVTYPE, ROOT, COMM, IERR)

Список параметров у этой подпрограммы похож на список параметров подпрограммы MPI_scatterv. Читателю не составит труда самостоятельно разобраться с их назначением.

В обменах, выполняемых подпрограммами MPI_Allgather и MPI_Alltoall, нет главного процесса. Детали отправки и приема важны для всех процессов, участвующих в обмене. Подпрограмма MPI_Allgather собирает данные от всех процессов и распределяет их всем процессам (рис. 5.3). Действие этой подпрограммы равносильно действию последовательности вызовов подпрограммы MPI_Gather, в каждом из которых в качестве главного используются разные процессы. Прием выполняется во всех задачах. Буфер приема последовательно заполняется сообщениями от всех процессов-отправителей.

int MPI_Allgather(void *sendbuf, int sendcount,

MPI_Datatype sendtype, void *rcvbuf, int rcvcount,

MPI_Datatype rcvtype, MPI_Comm comm)

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RCVBUF,

RCVCOUNT, RCVTYPE, COMM, IERR)

Входные параметры этой подпрограммы:

- sendbuf — начальный адрес буфера передачи;
- sendcount — количество элементов в буфере передачи;
- sendtype — тип передаваемых данных;
- rcvcount — количество элементов, полученных от каждого процесса;
- rcvtype — тип данных в буфере приема;
- coram — коммуникатор.

Выходным параметром является адрес буфера приема (rcvbuf). Блок данных, переданный от i-го процесса, принимается каждым процессом и размещается в i-м блоке буфера приема recvbuf.

Подпрограмма MPi_Alltoall пересылает данные по схеме "каждый — всем":

int MPI_Alltoall(void *sendbuf, int sendcount,

MPI_Datatype sendtype, void *rcvbuf, int rcvcount,

MPI_Datatype rcvtype, MPI_Comm comm)

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE,

RCVBUF, RCVCOUNT, RCVTYPE, COMM, IERR)

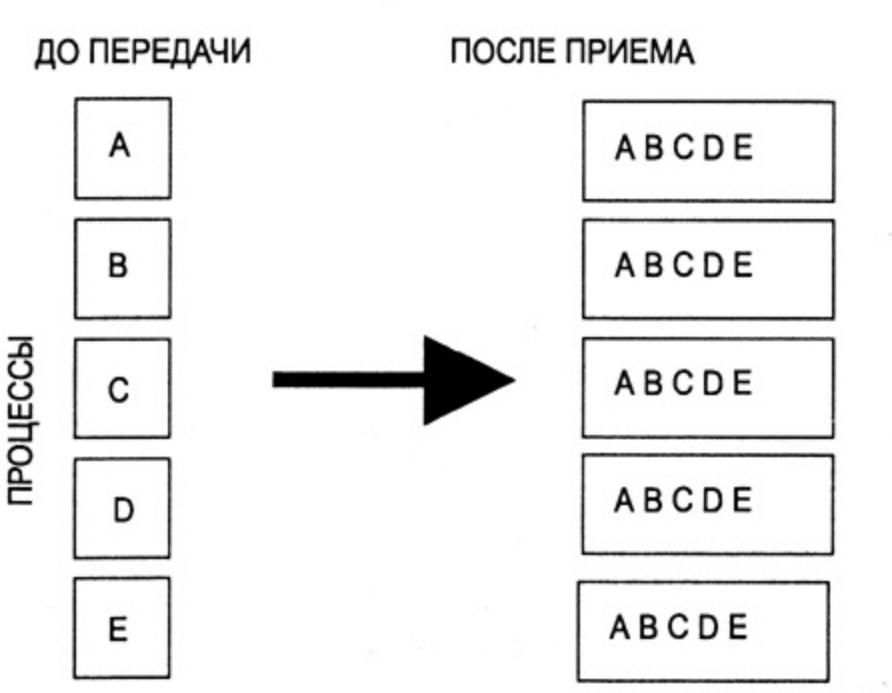


Рис. 5.3. *Схема передачи данных для подпрограммы MPI_Allgather*

Входные параметры:

- sendbuf — начальный адрес буфера передачи;
- sendcount — количество элементов данных, пересылаемых каждому процессу;
- sendtype — тип данных в буфере передачи;
- rcvcount — количество элементов данных, принимаемых от каждого процесса;
- rcvtype — тип принимаемых данных;
- comm — коммуникатор.

Выходной параметр — адрес буфера приема rcvbuf. Векторными версиями

MPI_Allgather и MPI_Alltoall является подпрограммы

MPI_Allgatherv и MPI_Alltoallv.

Подпрограмма MPI_Allgatherv собирает данные от всех процессов и пересылает их всем процессам:

int MPI_Allgatherv(void *sendbuf, int sendcount,

MPI_Datatype sendtype,

void *rcvbuf, int *rcvcounts, int *displs,

MPI_Datatype rcvtype, MPI_Comm

comm)

MPI_ALLGATHERV(SENDBUF, SENDCOUNT,

SENDTYPE, RCVBUF, RCVCOUNTS, DISPLS,

RCVTYPE, COMM, IERR)

Ее параметры совпадают с параметрами подпрограммы MPI_Allgatner, за исключением дополнительного входного параметра dispis. Это целочисленный одномерный массив, количество элементов в котором равно количеству процессов в коммуникаторе. Элемент массива с индексом i задает смещение относительно начала буфера приема recvbuf, в котором располагаются данные, принимаемые от процесса i. Блок данных, переданный от j-го процесса, принимается каждым процессом и размещается в y'-м блоке буфера приема.

Подпрограмма MPI_Mitoaliv пересылает данные от всех процессов всем процессам со смещением:

int MPI_Alltoallv(void *sendbuf, int *sendcounts,

int *sdispls, MPI_Datatype sendtype, void *rcvbuf,

int *rcvcounts, int *rdispls, MPI_Datatype rcvtype, MPI_Comm comm)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS,

SDISPLS, SENDTYPE, RCVBUF, RCVCOUNTS,

RDISPLS, RCVTYPE, COMM, IERR)

Ее параметры аналогичны параметрам подпрограммы MPi_Alltoall, кроме двух дополнительных параметров:

- sdispls — целочисленный массив, количество элементов в котором равно количеству процессов в коммуникаторе. Элемент *j* задает смещение относительно начала буфера, из которого данные передаются y'-му процессу.
- rdispls — целочисленный массив, количество элементов в котором равно количеству процессов в коммуникаторе. Элемент i задает смещение относительно начала буфера, в который принимается сообщение от i-го процесса.

Операции приведения и сканирования

Операции приведения и сканирования относятся к категории глобальных вычислений. В глобальной операции приведения к данным от всех процессов из заданного коммуникатора применяется операция MPI_Reduce (рис. 5.4).

Аргументом операции приведения является массив данных — по одному элементу от каждого процесса. Результат такой операции — единственное значение (поэтому она и называется операцией приведения).

В подпрограммах глобальных вычислений функция, передаваемая в подпрограмму, может быть: предопределенной функцией MPI, например MPI_SUM, пользовательской функцией, а также обработчиком для пользовательской функции, который создается подпрограммой Mpi_op_create.

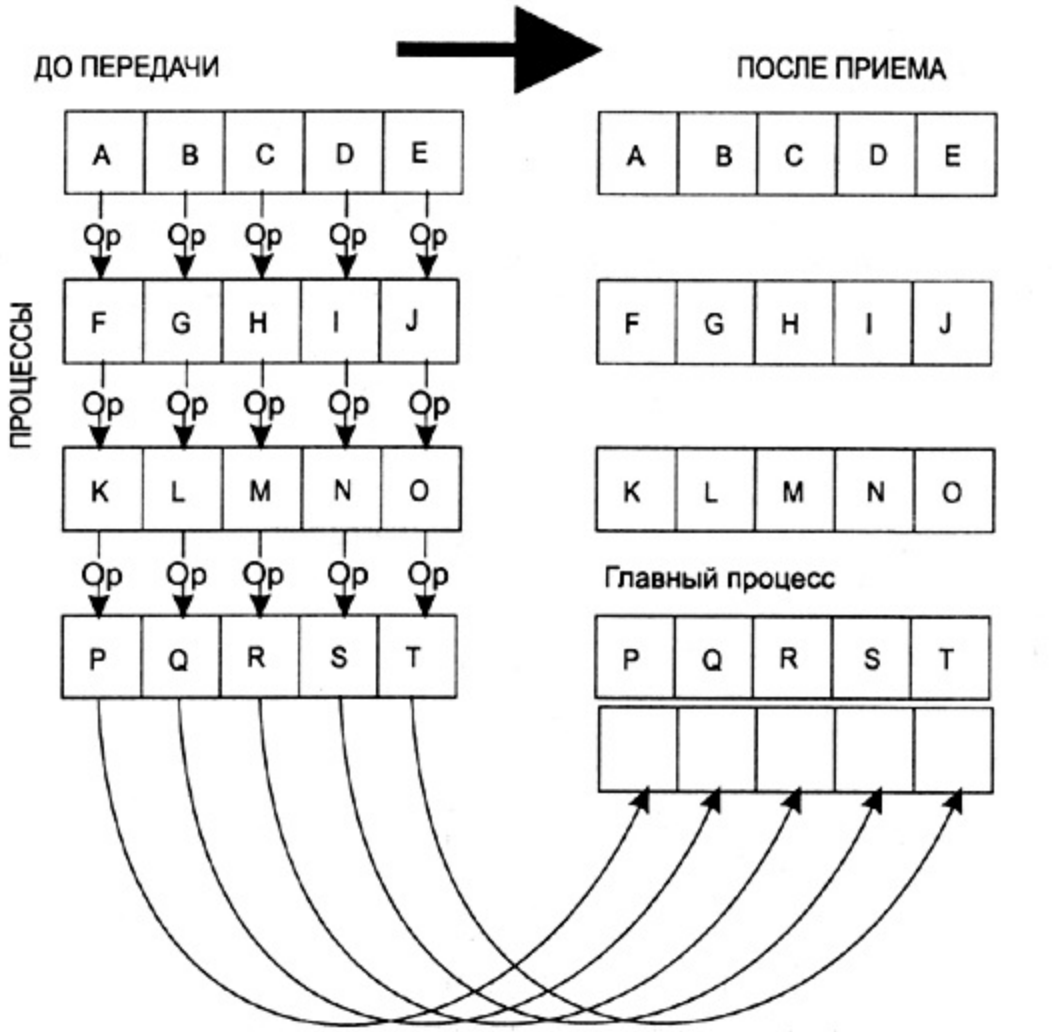


Рис. 5.4. *Глобальная операция приведения*

Три версии операции приведения возвращают результат:

- одному процессу;
- всем процессам;
- распределяют вектор результатов между всеми процессами.

Операция приведения, результат которой передается одному процессу, выполняется при вызове подпрограммы MPi_Reduce:

```
int MPI_Reduce(void *buf, void *result,
```

int count, MPI_Datatype datatype,

MPI_Op op, int root, MPI_Comm comm)

MPI_REDUCE(BUF, RESULT, COUNT,

DATATYPE, OP, ROOT, COMM, IERR)

Входные параметры подпрограммы MPi_Reduce:

- buf — адрес буфера передачи;
- count — количество элементов в буфере передачи;
- datatype — тип данных в буфере передачи;
- op — операция приведения;
- root — ранг главного процесса;
- comm — коммуникатор.

Подпрограмма MPi_Reduce применяет операцию приведения к операндам из buf, а результат каждой операции помещается в буфер результата result. MPi__Reduce должна вызываться всеми процессами в коммуникаторе comm, а аргументы count, datatype и op в этих вызовах должны совпадать. Функция приведения (op) не возвращает код ошибки, поэтому при возникновении аварийной ситуации либо завершается работа всей программы, либо ошибка молчаливо игнорируется. И то и другое в равной степени нежелательно.

В MPI имеется 12 предопределенных операций приведения (табл. 5.2).

Таблица 5.2. *Список предопределенных операций приведения MPI*

Операция	**Описание**
MPI_MAX	Определение максимальных значений элементов одномерных массивов целого или вещественного типа
MPI_MIN	Определение минимальных значений элементов одномерных массивов целого или вещественного типа
MPI_SUM	Вычисление суммы элементов одномерных массивов целого, вещественного или комплексного типа

MPI_PROD	Вычисление поэлементного произведения одномерных массивов целого, вещественного или комплексного типа
MPI LAND	Логическое "И"
MPI BAND	'Битовое "И"
MPI_LOR	Логическое "ИЛИ"
MPI_BOR	Битовое "ИЛИ"
MPI_LXOR	Логическое исключающее "ИЛИ"
MPI_BXOR	Битовое исключающее "ИЛИ"
MPI MAXLOC	Максимальные значения элементов одномерных массивов и их индексы
MPI MINLOC	Минимальные значения элементов одномерных массивов и их индексы

Программист может определить и собственные глобальные операции приведения. Это делается с помощью подпрограммы MPi_op_create:

int MPI_Op_create(MPI_User_function *function,

int commute, MPI_Op *op) MPI_OP_

CREATE(FUNCTION, COMMUTE, OP, IERR)

Входными параметрами этой подпрограммы являются:

- function — пользовательская функция;
- commute — флаг, которому присваивается значение "истина", если операция коммутативна (т. е. ее результат не зависит от порядка операндов).

Описание типа пользовательской функции выглядит следующим образом:

typedef void (MPI_User_function)

(void *a,* void *b, int *len, MPI_Datatype *dtype)

Здесь операция определяется так:

b[I] = *a [I]* op b[I] ДЛЯ 1 = 0, ..., len - *1.*

После того как пользовательская функция сыграла свою роль, ее следует аннулировать с помощью подпрограммы MPi_op_f гее:

int MPI_Op_free(MPI_Op *op) MPI_OP_FREE(OP, IERR)

По завершении вызова переменной op присваивается значение MPI_OP_NULL.

У подпрограммы MPi_Reduce есть варианты. Подпрограмма MPi_Reduce_scatter собирает и распределяет данные:

int MPI_Reduce_scatter(void *sendbuf, void *rcvbuf,

int *rcvcounts, MPI_Datatype datatype,

MPI_Op op, MPI_Comm comm)

MPI_REDUCE_SCATTER(SENDBUF, RCVBUF,

RCVCOUNTS, DATATYPE, OP, COMM, IERR)

Ее входные параметры:

- sendbuf — стартовый адрес буфера приема;
- rcvcounts — целочисленный одномерный массив, который задает количество элементов в результирующем массиве, распределяемом каждому процессу. Этот массив должен быть одинаковым во всех процессах, вызывающих данную подпрограмму;
- datatype — тип данных в буфере приема;
- op — операция;
- comm — коммуникатор.

Выходной параметр — стартовый адрес буфера приема rcvbuf. Особенностью этой подпрограммы является то, что каждая задача получает не весь результирующий массив, а его часть.

Подпрограмма MPi_Alireduce собирает данные от всех процессов и сохраняет результат операции приведения в результирующем буфере каждого процесса:

int MPI_Allreduce(void *sendbuf, void *rcvbuf, int count, MPI_Datatype

datatype, MPI_Op op, MPI_Comm comm)

MPI_ALLREDUCE(SENDBUF, RCVBUF, COUNT, DATATYPE, OP, COMM, IERR)

Ее входные параметры:

- sendbuf — начальный адрес буфера передачи;
- count — количество элементов в буфере передачи;
- datatype — тип передаваемых данных;
- op — операция приведения;
- comm **—** коммуникатор**.**

Выходным параметром является стартовый адрес буфера приема rcvbuf. При аварийном завершении подпрограмма может возвращать код ошибки MPI_ERR_OP (некорректная операция). Это происходит, если применяется операция, которая не является предопределенной и которая не создана предшествующим вызовом подпрограммы MPi_op_create.

Операции сканирования (частичной редукции) выполняются с помощью вызова подпрограммы MPI_Scan:

int MPI_Scan(void *sendbuf, void *rcvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) MPI_SCAN(SENDBUF, RCVBUF, COUNT, DATATYPE, OP, COMM, IERR)

Ее входные параметры:

- sendbuf — начальный адрес буфера передачи;
- ount — количество элементов во входном буфере;
- datatype — тип данных во входном буфере;
- op — операция;
- comm — коммуникатор.

Выходным параметром является стартовый адрес буфера приема rcvbuf. Схема распределения данных при выполнении операции сканирования приведена на рис. 5.5.

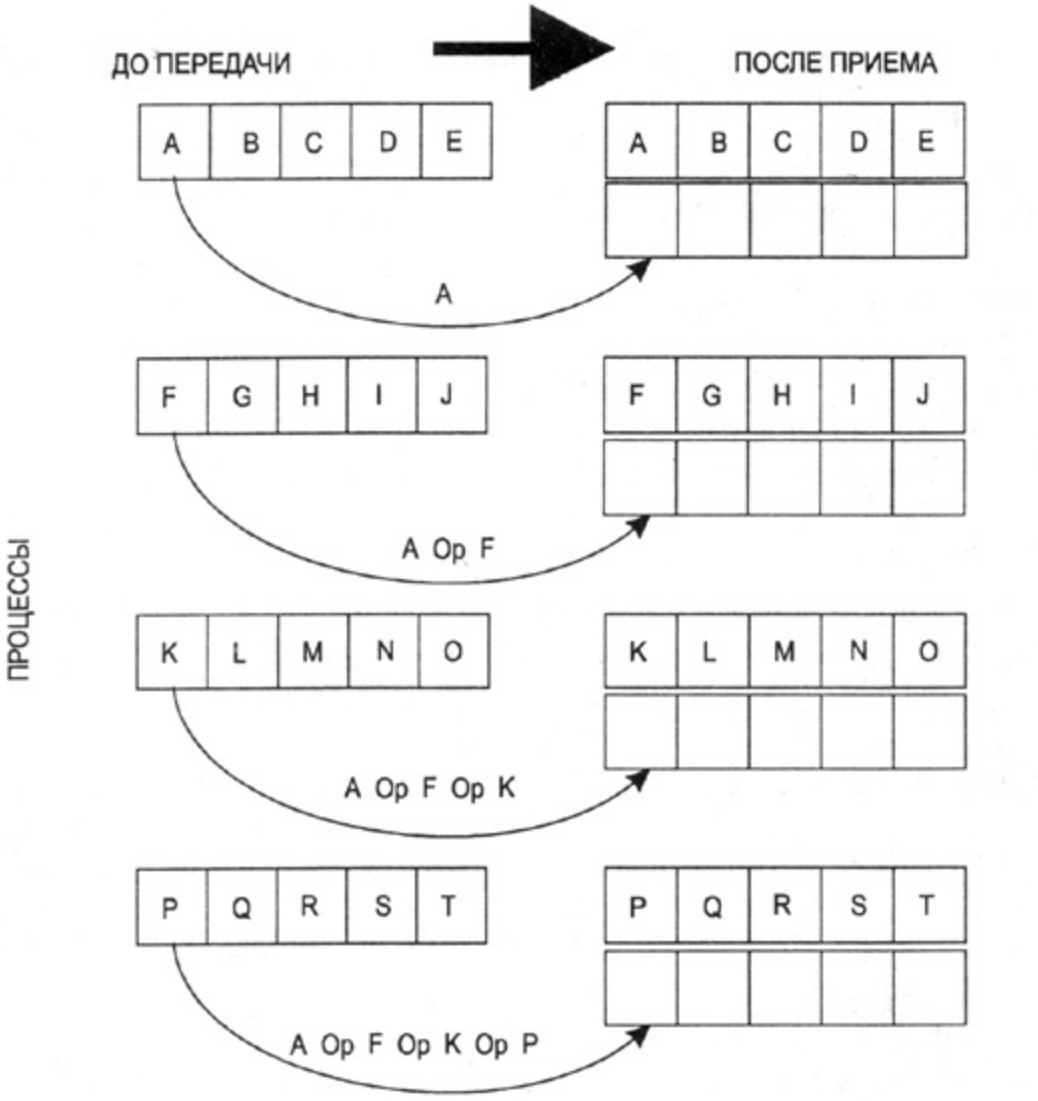


Рис. 5.5. *Операция сканирования*

В подпрограмме MPi_Reduce можно использовать только предопределенные типы MPI. Все операции являются ассоциативными и коммутативными (т. е. операнды могут по-разному группироваться, а их порядок не имеет значения). Подпрограмма MPI_scan похожа на подпрограмму MPi_Allreduce в том отношении, что каждая задача получает результирующий массив. Отличие состоит в том, что содержимое массива-результата в задаче i является результатом выполнения операции над массивами из задач с номерами от 0 до i включительно.

В каждом процессе можно использовать разные пользовательские операции. В MPI не определено, какие операции и над какими операндами будут применяться в этом случае. В буферах передачи допускается перекрытие типов, в буферах приема это может привести к непредсказуемым результатам.

Пример использования редукции совместно с двухточечными обменами приведен в листинге 5.2.

Листинг 5.2. Пример использования операции редукции

```
#include "mpi.h"
finclude <stdio.h>
int main(int argc,char *argv[])
{
int rayrank, i;
int count = 5, root = 1;
MPI_Group MPI_GROUP_WORLD, subgroup;
int ranks[4] = {1, 3, 5, 7};
MPI_Comm subcomm;
int sendbuf[5] = {1, 2, 3, 4, 5};
int recvbuf[5];
MPI_Init(sargc, sargv);
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, Ssubgroup);
MPI_Group_rank(subgroup, Smyrank);
MPI_Coram_create(MPI_COMM_WORLD, subgroup, ssubcomm);
if(myrank != MPIJJNDEFINED)
{
MPI_Reduce(&sendbuf, &recvbuf, count, MPI_INT, MPI_SUM, root, subcomm);
if(myrank == root) { printf("Reduced values");
for(i = 0; i < count; i++){ printf(" %i ", recvbuf[i]);}
printf ("\n") ;
MPI_Comm_free (&subcomm) ;
MPI_Group_free (&MPI_GROUP_WORLD)
```

```
MPI_Group_free (&subgroup) ; }

MPI_Finalize() ;

return 0;

}
```

В этой программе сначала создается подгруппа, состоящая из процессов с рангами 1, 3, 5 и 7 (отсюда следует, что запускать ее на выполнение надо не менее чем в восьми процессах), и соответствующий ей коммуникатор. Редукция выполняется только процессами из этой группы. В конце программы все созданные в процессе ее работы описатели должны быть удалены.

Топологии

Овладев искусством организации двухточечных и коллективных обменов, познакомимся с некоторыми дополнительными возможностями **MPI,** которые позволяют в определенных ситуациях упростить разработку параллельных программ. Организация коллективного обмена, как мы уже знаем, основана на свойствах коммуникатора — описателя области взаимодействия. Кроме списка процессов и контекста обмена с коммуникатором может быть связана дополнительная информация. Говорят, что она *кэшируется* с коммуникатором. Важнейшей разновидностью такой информации является *топология* обменов. В MPI топология представляет собой механизм сопоставления процессам, принадлежащим группе, альтернативных по отношению к обычной схем адресации. Топологии обменов сообщениями в MPI являются виртуальными, они не связаны с физической топологией коммуникационной сети параллельной вычислительной системы. Использование коммуникаторов и топологий отличает MPI от большинства других систем передачи сообщений.

Читателю, не знакомому с высшей математикой (а "топология" — математический термин), поясним, что топологией в данном случае называют структуру соединений линий и узлов сети без учета характеристик самих этих узлов. Узлами здесь являются процессы, соединениями — каналы обмена сообщениями, а сетью мы, фактически, называем все процессы, входящие в состав параллельной программы. Часто в прикладных программах процессы упорядочены в соответствии с определенной топологией. Такая ситуация возникает, например, если выполняются расчеты, в которых используются решетки (или "сетки"). Это может быть при программировании сеточных методов решения дифференциальных уравнений, а также в других случаях.

Знание топологии задачи можно использовать для того, чтобы эффективно распределить процессы между процессорами параллельной вычислительной системы.

В MPI существуют два типа топологии: декартова топология — прямоугольная решетка произвольной размерности (рис. 5.6) и топология графа (в этом случае процессы соединены между собой ребрами, показывающими направление обмена — рис. 5.7).

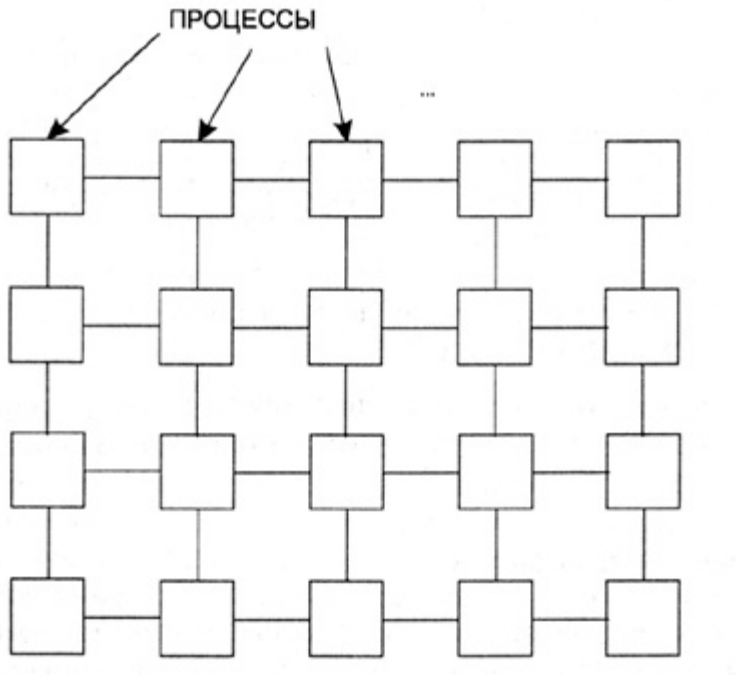


Рис. 5.6. *Декартова топология*

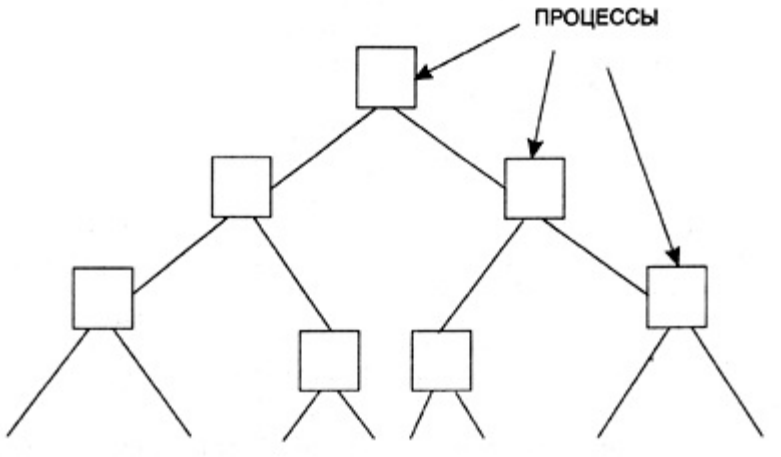


Рис. 5.7. *Топология графа*

Над топологиями можно выполнять различные операции. Декартовы решетки, например, можно расщеплять на гиперплоскости, удаляя некоторые измерения. Данные можно сдвигать вдоль выбранного измерения декартовой решетки. *Сдвигом* в этом случае называют пересылку данных между процессами вдоль определенного измерения. Вдоль избранного измерения могут быть организованы коллективные обмены.

Для того чтобы связать структуру декартовой решетки с коммуникатором MPI_COMM_WORLD, необходимо задать следующие параметры:

- размерность решетки (значение. 2, например, соответствует плоской, двумерной решетке);
- размер решетки вдоль каждого измерения (размеры {10, 5}, например, соответствуют прямоугольной плоской решетке, дротяженность которой вдоль оси *x* составляет 10 узлов-процессов, а вдоль оси *y* — 5 узлов);
- периодичность вдоль каждого измерения (решетка может быть периодической, если процессы, находящиеся на противоположных концах ряда, взаимодействуют между собой).

MPI дает возможность системе оптимизировать отображение виртуальной топологии процессов на физическую с помощью изменения порядка нумерации процессов в группе.

Декартовы топологии

Познакомимся с операциями создания и преобразования *декартовых топологий* обмена. Подпрограмма MPI_cart_create создает новый коммуникатор comm_cart, наделяя декартовой топологией исходный коммуникатор comm_old:

int MPI_Cart_create(MPI_Coram comm_old,

int ndims, int *dims, int *periods, int reorder,

MPI_Comm *comm_cart) MPI_CART_

CREATE(COMM_OLD, NDIMS, DIMS,

PERIODS, REORDER, COMM_CART, IERR)

Входные параметры:

П comm_old — исходный коммуникатор;

- ndims — размерность декартовой решетки;
- dims — целочисленный массив, состоящий из ndims элементов, задающий количество процессов в каждом измерении;
- periods — логический массив из ndims элементов, который определяет, является ли решетка периодической (значение "истина") вдоль каждого измерения;
- reorder - при значении этого параметра "истина" системе разрешено менять порядок нумерации процессов.

Информация о структуре декартовой топологии содержится в параметрах ndims, dims И periods. MPI_Cart_create является коллективной операцией (эту подпрограмму должны вызывать все процессы из коммуникатора, наделяемого декартовой

топологией).

После создания виртуальной топологии можно использовать соответствующую схему адресации процессов, но для этого требуется пересчет ранга процесса в его декартовы координаты и наоборот. Определить декартовы координаты процесса по его рангу в группе можно с помощью подпрограммы MPI_Cart_coords:

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERR)

Входные параметры:

- comm — коммуникатор, наделенный декартовой топологией;
- rank — ранг процесса в comm;
- maxdims — длина вектора coords в вызывающей программе.

Выходным параметром этой подпрограммы является одномерный целочисленный массив coords (его размер равен ndims), содержащий декартовы координаты процесса.

Обратным действием по отношению к MPi_cart_coords обладает подпрограмма MPI_Cart_rank. С ее помощью можно определить ранг процесса (rank) по его декартовым координатам в коммуникаторе comm:

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

MPI_CART_RANK(COMM, COORDS, RANK, IERR)

Входной параметр coords — целочисленный массив размера ndims, задающий декартовы координаты процесса. Как MPi_Cart_rank, так и MPi_cart_coords локальны.

Пример создания декартовой топологии приведен в листинге 5.3.

Листинг 5.3. Пример создания декартовой топологии

```
#include "mpi.h"

#include <stdio.h>-

int main (int argc,char *argv[])

}

MPI_Comm grid_comm;

int dims [ 2 ] ;
```

```
int periodic[2];

int reorder =1, q = 5, ndims = 2, maxdims = 2;

int coordinates [2] ;

int my_grid_rank; int coords [2] ;

MPI_Comm row_coram;

dims[0] = dims[l] = q; periodic [0] = periodic [1] = 1;

coords [ 0 ] = 0 ; coords [ 1 ] = 1 ; MPI__Init (&argc, sargv) ;

MPI_Cart_create (MPI_COMM_WORLD, ndims,

dims, periodic, reorder, &grid_comm) ;

MPI_Comm_rank(grid_comm, &my_grid_rank) ;

MPI_Cart_coords (grid_comm, my_grid_rank, maxdims, coordinates);

printf ("Process rank %i has coordinates %i %i\n",

my_grid_rank, coordinates [0], coordinates [1] );

MPI_Finalize() ; return 0 ;

}
```

В этой программе коммуникатор grid_comm наделяется топологией двумерной решетки с периодическими граничными условиями, причем системе разрешено изменить порядок нумерации процессов. С учетом последнего, каждый процесс из коммуникатора grid_comm может определить свой ранг с помощью подпрограммы MPI_comm_rank. Декартовы координаты определяются при вызове подпрограммы MPI_cart_coords. Результат выполнения этой программы выглядит так:

```
# mpirun -np 25 a. out

Process rank 0 has coordinates 0 0

Process rank 1 has coordinates 0 1

Process rank 2 has coordinates 0 2

Process rank 3 has coordinates 0 3
```

Process rank 4 has coordinates 0 4

Process rank 5 has coordinates 1 0

Process rank 6 has coordinates 1 1

Process rank 7 has coordinates 1 2

Process rank 8 has coordinates 1 3

Process rank 9 has coordinates 1 4

Process rank 10 has coordinates 2 0

Process rank 11 has coordinates 2 1

Process rank 12 has coordinates 2 2

Process rank 13 has coordinates 2 3

Process rank 14 has coordinates 2 4

Process rank 15 has coordinates 3 0

Process rank 16 has coordinates 3 1

Process rank 17 has coordinates 3 2

Process rank 18 has coordinates 3 3

Process rank 19 has coordinates 3 4

Process rank 20 has coordinates 4 0

Process rank 21 has coordinates 4 1

Process rank 22 has coordinates 4 2

Process rank 23 has coordinates 4 3

Process rank 24 has coordinates 4 4

Для наглядности порядок строк вывода здесь изменен. Заметим, что в программе создается топология двумерной квадратной решетки 5x5, поэтому количество процессов при запуске должно быть 25. Читатель в качестве упражнения может попробовать добавить в эту программу обмен сообщениями, в котором явным образом учитывается декартова топология процессов.

Подпрограмма MPI_cart_sub расщепляет коммуникатор comm на подгруппы, соответствующие декартовым подрешеткам меньшей размерности:

int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,

MPI_Comm *comm_new) MPI_CART_

SUB(COMM, REMAIN_DIMS, COMM_NEW, IERR)

Здесь i-й элемент массива remain_dims определяет, содержится ли i-е измерение в подрешетке ("истина"). Выходным параметром является коммуникатор newcomm, содержащий подрешетку, которой принадлежит данный процесс.

Пример разбиения двумерной решетки на одномерные подрешетки, соответствующие ее рядам, приводится ниже:

int coords[2] ;

MPI_Corara row_comm;

coords[0] = 0; coords[1] = 1;

MPI_Cart_sub(grid_comm, coords, &row_comm);

В этом примере вызов подпрограммы MPi_cart_sub создает несколько новых коммуникаторов. Аргумент coords определяет, принадлежит ли соответствующее измерение новому коммуникатору.

Несмотря на то, что подпрограмма MPI_Cart_sub выполняет-функции, аналогичные MPI_comm_split, т. е. расщепляет коммуникатор на набор новых коммуникаторов, между ними есть существенное различие — подпрограмма MPI_Cart_sub используется только с коммуникатором, наделенным декартовой топологией.

Информацию о декартовой топологии, связанной с коммуникатором comm, МОЖНО ПОЛУЧИТЬ С ПОМОЩЬЮ подпрограммы MPI_Cart_get:

int MPI_Cart_get(MPI_Cornm comm, int maxdims, int *dims, int *periods,

int *coords)

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERR) ВХОДНОЙ параметр maxdims задает длину массивов dims, periods И vectors в вызывающей программе, а выходные параметры:

- dims - целочисленный массив, задающий количество процессов для каждого

измерения;
- periods — массив логических значений, задающих периодичность ("истина", если решетка периодическая) для каждого измерения;
- coords — целочисленный массив, задающий декартовы координаты вызывающего подпрограмму процесса.

С помощью подпрограммы MPI_Cart_map можно определить ранг процесса (newrank) в декартовой топологии после переупорядочения процессов:

int MPI_Cart_map(MPI_Comm comm_old, int ndims, int *dims, int ^periods,

int *newrank)

MPI_CART_MAP(COMM_OLD, NDIMS,

DIMS, PERIODS, NEWRANK, IERR)

Входные параметры:

- comm — коммуникатор;
- ndims — размерность декартовой решетки;
- dims — целочисленный массив, состоящий из ndims элементов, который определяет количество процессов вдоль каждого измерения;
- periods — логический массив размера ndims, определяющий периодичность решетки вдоль каждого измерения.

Если процесс не принадлежит решетке, подпрограмма возвращает значение

MPI_UNDEFINED.

Между процессами, организованными в декартову решетку, могут выполняться обмены особого вида. Это сдвиги, о которых мы уже упоминали. Имеются два типа сдвигов данных по группе из *N* процессов:

- *циклический сдвиг* на J позиций вдоль ребра решетки. Данные от процесса *K* пересылаются процессу (*J + K)* mod *N;*
- *линейный сдвиг* на J позиций вдоль ребра решетки, когда данные в процессе *K* пересылаются процессу J + *K,* если ранг адресата находится в пределах между 0 и *N.*

Ранги источника (source) сообщения, которое должно быть принято, и адресата (dest), который должен получить сообщение для заданного направления сдвига (direction) и его величины (disp), можно определить с помощью подпрограммы MPI_Cart_shift:

int MPI_Cart_shift(MPI_Comm comm, int direction, int displ,

int *source, int *dest)

MPI_CART_SHIFT(COMM, DIRECTION, DISPL, SOURCE, DEST, IERR)

Для n-мерной декартовой решетки значение аргумента direction должно находиться в пределах от 0 до *п* — 1.

Подпрограмма MPi_Cartdim_get позволяет определить размерность (ndims) декартовой топологии, связанной с коммуникатором comm:

int MPI_Cartdim_get(MPI_Comm comm, int *ndims) MPI_CARTDIM_GET(COMM, NDIMS, IERR)

Топология графа

Мы ограничимся обзором основных подпрограмм MPICH, предназначенных для работы с топологией графа. Подпрограмма MPI_Graph_create создает новый коммуникатор comm_graph, наделенный топологией графа:

int MPI_Graph_create(MPI_Comm comm, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph)

MPI_GRAPH_CREATE(COMM, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, IERR)

Входные параметры:

- comm — "исходный" коммуникатор, не наделенный топологией;
- nnodes — количество вершин графа;
- index — целочисленный одномерный массив, содержащий порядок каждого узла (количество связанных с ним ребер);
- edges — целочисленный одномерный массив, описывающий ребра графа;
- reorder — значение "истина" разрешает изменение порядка нумерации процессов.

Подпрограмма Mpi_Graph_neighbors возвращает соседние с данной вершины графа:

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,

int *neighbors)

MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERR)

Ее входные параметры:

- comm — коммуникатор с топологией графа;
- rank — ранг процесса в группе коммуникатора comm;
- maxneighbors — размер массива neighbors.

Выходным параметром является массив neighbors, содержащий ранги процессов, соседних с данным.

Количество соседей (nneighbors) узла, связанного с топологией графа, можно определить с помощью подпрограммы MPI_Graph_neighbors_count:

int MPI_Graph_neighbors_count

(MPI_Comm comm, int rank, int *nneighbors)

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERR)

Входные параметры:

- comm — коммуникатор;
- rank — ранг процесса-узла.

Получить информацию о топологии графа, связанной с коммуникатором

comm, можно с помощью подпрограммы MPI_Graph_get:

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,

int *edges)

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERR)

Входными параметрами являются:

- comm — коммуникатор;
- maxindex — длина массива index в вызывающей программе;
- maxedges — длина массива edges в вызывающей программе.

Выходные параметры:

- index — целочисленный массив, содержащий структуру графа (см. описание подпрограммы MPI_Graph_create);
- edges — целочисленный массив, содержащий сведения о ребрах графа.

С помощью подпрограммы MPI_Graph_map можно определить ранг процесса в топологии графа после переупорядочения (newrank):

int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,

int *newrank)

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERR)

Ее входные параметры:

- comm — коммуникатор;
- nnodes — количество вершин графа;
- index — целочисленный массив, задающий структуру графа (см. описание подпрограммы MPI_Graph_create);
- edges — целочисленный массив, задающий ребра графа.

Если процесс не принадлежит графу, подпрограмма возвращает значение

MPI UNDEFINED.

Подпрограмма MPI_Graphdims_get позволяет получить информацию о топологии графа, связанной с коммуникатором comm:

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERR)

Выходными параметрами этой подпрограммы являются:

- nnodes — количество вершин графа;
- nedges — количество ребер графа.

И, наконец, определить тип топологии (toptype), связанной с коммуникатором comm, можно с помощью подпрограммы MPI_Topo_test:

int MPI_Topo_test(MPI_Comm comm, int *toptype)

MPI_TOPO_TEST(COMM, TOPTYPE, IERR)

Выходным параметром является тип топологии toptype (значения MPI_CART для декартовой топологии и MPI_GRAPH для топологии графа).

Производные типы данных

Сообщение в MPI представляет собой массив однотипных данных, элементы которого расположены в последовательных ячейках памяти (в этом случае говорят о непрерывном расположении элементов сообщения). Такая структура не всегда удобна. Иногда возникает необходимость в пересылке разнотипных данных или

фрагментов массивов, содержащих элементы, расположенные не в последовательных ячейках. Так бывает, например, при программировании вычислений, связанных с операциями с матрицами и векторами, а также в других ситуациях.

Приведем простой пример. В численных расчетах часто приходится иметь дело с матрицами. Математик знает, что это такое, а программист может считать, что матриць! представляют собой двумерные массивы, содержащие численные значения, к которым могут применяться специальные операции, такие как сложение и умножение, вычисление обратной матрицы и т. д. В языке FORTRAN (и С) используется линейная модель памяти, в которой матрица хранится в виде последовательно расположенных столбцов (строк). Один из алгоритмов параллельного умножения матриц требует пересылки строк матрицы, а в линейной модели элементы строк расположены в оперативной памяти не непрерывно, а с промежутками (рис. 5.8).

Решить проблему пересылки разнотипных данных или данных, расположенных не в последовательных ячейках памяти, можно уже известными нам средствами. Можно, например, "уложить" элементы исходного массива во вспомогательный массив так, чтобы данные располагались непрерывно. Но это не очень удобно и, к тому же, требует дополнительных затрат памяти и процессорного времени. Можно различные элементы данных пересылать по отдельности. Это тоже чрезвычайно медленный и неудобный способ обмена. Более эффективным решением является использование *производных типов данных.*

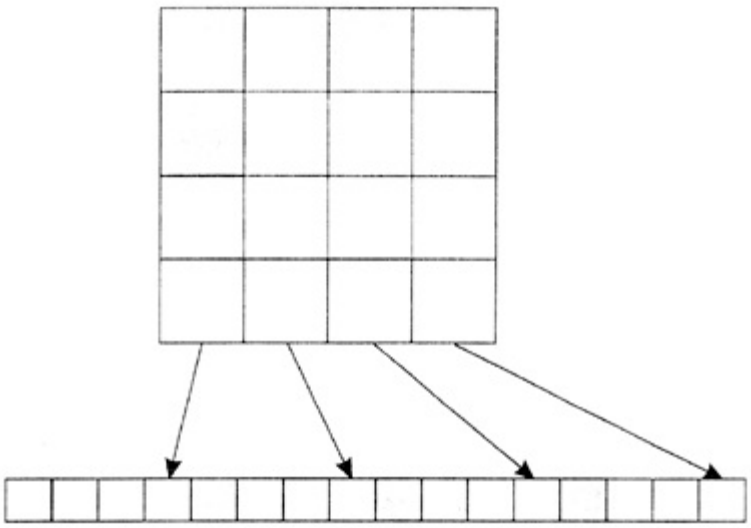


Рис. 5.8. *Расположение элементов строки матрицы в линейной модели памяти*

Производные типы данных создаются во время выполнения программы (а не во время ее трансляции), как правило, перед их использованием. Создание типа — двухступенчатый процесс, который состоит из двух шагов:

1. Конструирование типа.

2. Регистрация типа.

После завершения работы с производным типом он аннулируется. При этом все производные от него типы остаются и могут использоваться дальше, пока и они не будут уничтожены. Здесь уместно напомнить, что коммуникаторы, группы, типы данных, распределенные операции, запросы, атрибуты коммуникаторов, обработчики ошибок — все это объекты, которые (если они были созданы в процессе работы программы) должны удаляться. Последовательность удаления может быть любой.

Производные типы данных создаются из базовых типов с помощью подпрограмм-конструкторов. Операции создания производных типов могут применяться рекурсивно, что дает программисту большую свободу в организации сложных типов данных.

Производный тип данных в MPI характеризуется последовательностью базовых типов и набором целочисленных значений смещения. Смещения отсчитываются относительно начала буфера обмена и определяют те элементы данных, которые будут участвовать в обмене. Смещения могут принимать как положительные, так и отрицательные значения. Не требуется также, чтобы они были упорядочены (например, по возрастанию или по убыванию). Отсюда следует, что порядок элементов данных в производном типе может отличаться от исходного и, кроме того, один элемент данных может появляться в новом типе многократно. Последовательность пар (тип, смещение) называется *картой типа* (табл. 5.3).

Таблица 5.3. *Карта производного типа данных*

1-й элемент пары	**2-й элемент пары**
Базовый тип 0	Смещение базового типа 0
Базовый тип 1	Смещение базового типа 1
Базовый тип 2	Смещение базового типа 2
Базовый тип n — 1	Смещение базового типа n - 1

Обратим внимание на то, что расстояние в количестве ячеек задается между началами элементов, поэтому элементы могут и располагаться с разрывами, и перекрываться между собой.

Конструкторы производных типов

Подпрограмма MPI_Type_struct является наиболее общим конструктором типа в MPI, поэтому программист может использовать полное описание каждого элемента типа. Если пересылаемые данные содержат подмножество элементов массива такая детальная информация не нужна, поскольку у всех элементов один и тот же базовый тип. MPI содержит три конструктора, которые можно использовать в такой ситуации: MPI_Type_contiguous, MPI_Type_vector и MPI_Type_indexed. Первый из них создает производный тип, элементы которого являются непрерывно расположенными элементами массива. Второй создает тип, элементы которого находятся на одинаковых расстояниях друг от друга, а третий создает тип, содержащий произвольные элементы.

"Векторный" тип создается конструктором MPi_Type_vector:

int MPI_Type_vector(int count, int blocklen, int stride,

MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI TYPE VECTOR(COUNT, BLOCKLEN, STRIDE, OLDTYPE, NEWTYPE, IERR)

Входные параметры:

- count — количество блоков (неотрицательное целое значение);
- blocklen — длина каждого блока (количество элементов, неотрицательное целое);
- stride — количество элементов, расположенных между началом предыдущего и началом следующего блока ("гребенка");
- oldtype — базовый тип.

Выходным параметром является идентификатор нового типа newtype. Этот идентификатор назначается программистом. Исходные данные здесь однотипные. Схема расположения данных в новом типе представлена на рис. 5.9.

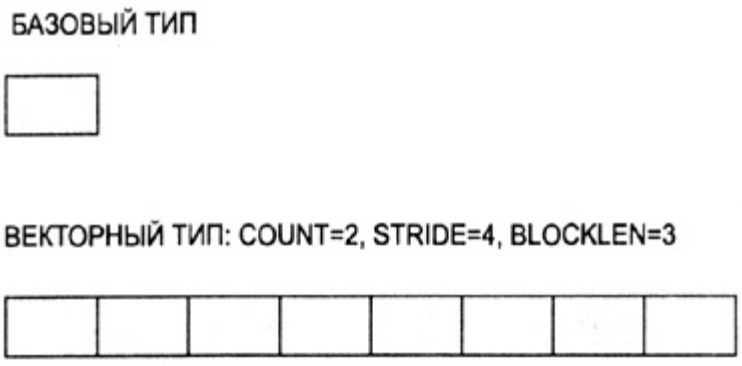


Рис. 5.9. *Схема векторного типа данных*

Если, например,

count = 2; stride = 4; biockien = 3;

a

oldtype = double;

то карта нового типа newtype будет выглядеть так:

{(double, 0), (double, 1), (double, 2),

(double, 4), (double, 5), (double, 6) }

При вызове подпрограммы MPi_Type_hvector смещения задаются в байтах:

int MPI_Type_hvector(int count, int biockien, MPI_Aint stride,

MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_HVECTOR(COUNT, BLOCKLEN,

STRIDE, OLDTYPE, NEWTYPE, IERR)

Смысл и назначение параметров этой подпрограммы совпадают с подпрограммой MPI_Type_vector, только значение параметра stride задается в байтах.

Конструктором структурного типа является подпрограмма MPi_Type_struct. Она позволяет создать тип, содержащий элементы различных базовых типов:

int MPI_Type_struct(int count, int blocklengths[],

MPI_Aint indices[], MPI_Datatype oldtypes[],

MPI_Datatype *newtype)

MPI_TYPE_STRUCT(COUNT, BLOCKLENGTHS,

INDICES, OLDTYPES, NEWTYPE, IERR)

Ее входные параметры:

- count — задает количество элементов в производном типе, а также длину массивов oldtypes, indices и blocklengths;
- blocklengths — количество элементов в каждом блоке (массив);
- indices — смещение каждого блока в байтах (массив);
- oldtypes — тип элементов в каждом блоке (массив).

Выходной параметр — идентификатор производного типа newtype. Схема расположения данных в структурном типе представлена на рис. 5.10. Данную подпрограмму можно использовать рекурсивно.

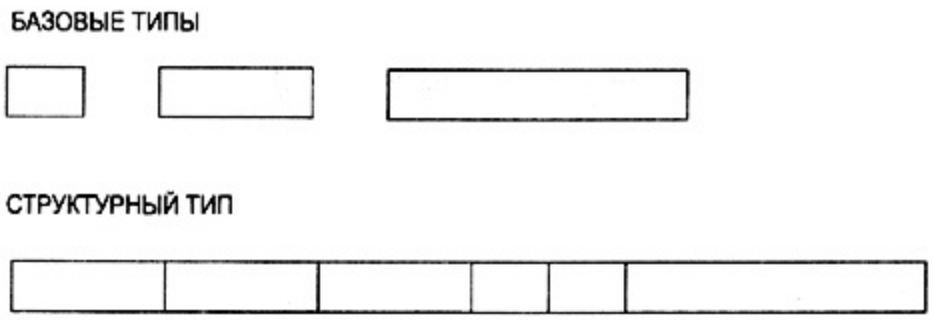


Рис. 5.10. *Схема структурного типа данных*

MPI_Aint представляет собой скалярный тип, длина которого имеет размер, одинаковый с указателем.

Пример создания структурного производного типа:

```
blen[0] = 1;

indices[0] = 0;

oldtypes[0] = MPI_INT;

blen[l] = 1;

indices[1] = &data.b — Sdata;

oldtypes[1] = MPI_CHAR;

blen[2] = 1;

indices[2] = sizeof(data);

oldtypes[2] = MPI_FLOAT;

MPI_Type_struct(3, blen, indices, oldtypes, Snewtype);
```

В качестве упражнения попробуйте построить карту нового типа newtype для этого примера. Достаточно ли для этого информации, содержащейся в приведенном фрагменте программы?

При создании индексированного типа блоки располагаются по адресам с разным смещением и его можно считать обобщением векторного типа. Конструктором

индексированного типа является подпрограмма MPi_Type_ indexed:

int MPI_Type_indexed(int count, int blocklens[], int indices[],

MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_INDEXED(COUNT, BLOCKLENS, INDICES, OLDTYPE, NEWTYPE, IERR)

Ее входные параметры:

- count - количество блоков, одновременно длина массивов indices и blocklens;
- blocklens — количество элементов в каждом блоке;
- indices — смещение каждого блока, которое задается в количестве ячеек базового типа (целочисленный массив);
- oldtype — базовый тип.

Выходным параметром является идентификатор производного типа newtype.

В языке FORTRAN роль смещений могут играть значения индексов массива. Смещение в этом случае отсчитывается относительно первого элемента массива. Численные значения смещений в картах типа отсчитываются от нуля, а значения индексов в FORTRAN по умолчанию отсчитываются от единицы. Для согласования обеих систем отсчета иногда удобно описывать массивы следующим образом:

REAL A(0:49)

Тогда значение индекса массива равно значению смещения.

Подпрограмма MPI_Type_hindexed также является конструктором индексированного типа, однако смещения indices задаются в байтах:

int MPI_Type_hindexed(int count, int blocklens[], MPI_Aint indices[],

MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_HINDEXED(COUNT, BLOCKLENS,

INDICES, OLDTYPE, NEWTYPE, IERR)

Подпрограмма MPI_Type_contiguous используется для создания типа данных с непрерывным расположением элементов:

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,

MPI_Datatype *newtype)

MPI TYPE CONTIGUOUS(COUNT,

OLDTYPE, NEWTYPE, IERR)

Ее входные параметры:

- count — счетчик повторений;
- oldtype — базовый тип.

Выходной параметр newtype — идентификатор нового типа. Эта подпрограмма фактически создает описание массива. Приведем фрагмент программы с ее использованием:

MPI_Datatype a; float b[10];

MPI_Type_contiguous(10, MPI_REAL, &a);

MPI_Type_commit (&a) ;

MPI_Send(b, I, a,...);

MPI_Type_free(&a);

Подпрограмма MPI_Type_create_indexed_block используется для создания индексированного типа с блоками постоянного размера:

int MPI_Type_create_indexed_block(int count, int blocklength, int displacements[],

MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, DISPLACEMENTS, OLDTYPE, NEWTYPE, IERR)

Ее входные параметры:

- count — количество блоков и размер массивов indices и biockiens;
- blocklength — количество элементов в каждом блоке;
- displacements — смещение каждого блока в единицах длины типа oldtype (целочисленный массив);
- oldtype — базовый тип.

Идентификатор производного типа newtype является выходным параметром этой подпрограммы.

Тип данных, соответствующий подмассиву многомерного массива, можно создать с помощью подпрограммы MPI_Type_create_subarray:

int MPI_Type_create_subarray(int ndims, int *sizes, int *subsizes, int *starts, int order,

MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_CREATE_S0BARRAY(NDIMS, SIZES, SUBSIZES,

STARTS, ORDER, OLDTYPE, NEWTYPE, IERR)

Входные параметры:

- ndims — размерность массива;
- sizes — количество элементов типа oldtype в каждом измерении полного массива;
- subsizes — количество элементов типа oldtype в каждом измерении под-массива;
- starts — стартовые координаты подмассива в каждом измерении;
- order — флаг, задающий переупорядочение;
- oidtype — базовый тип.

Новый тип newtype является выходным параметром этой подпрограммы.

Существуют и другие конструкторы производных типов, которые, впрочем, используются значительно реже.

Регистрация и удаление производных типов

С помощью вызова подпрограммы MPi_Type_commit производный тип datatype, сконструированный программистом, регистрируется. После этого он может использоваться в операциях обмена:

int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(DATATYPE, IEKR)

Аннулировать производный тип datatype можно с помощью вызова подпрограммы

MPI_Type_freel

int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE(DATATYPE, IERR)

Предопределенные (базовые) типы данных не могут быть аннулированы.

Вспомогательные подпрограммы

Кроме конструкторов при работе с производными типами иногда применяют различные вспомогательные подпрограммы.

Размер типа datatype в байтах (размер — это объем памяти, занимаемый одним элементом данного типа) можно определить с помощью вызова подпрограммы MPI_Type_size:

int MPI_Type_size(MPI_Datatype datatype, int *size)

MPI_TYPE_SIZE(DATATYPE, SIZE, IERR)

Выходным параметром является размер size.

Количество элементов данных в одном объекте типа datatype (его *экстент)* можно определить с помощью вызова подпрограммы MPi_Type_extent:

int MPI_Type_extent(MPI_Datatype datatype,

MPI_Aint *extent) MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERR)

Выходной параметр — extent.

Смещения могут даваться относительно базового адреса, значение которого содержится в константе MPI_BOTTOM.

Адрес (address) по заданному положению (location) можно определить с помощью подпрограммы MPI_Address:

int MPI_Address(void *location, MPI_Aint *address)

MPI_ADDRESS(LOCATION, ADDRESS, IERR)

Эта подпрограмма может использоваться в программах на языках C и FORTRAN. В C она обычно возвращает тот же адрес, что и оператор &, хотя иногда это и не так. Данная подпрограмма может понадобиться в программе на языке FORTRAN, а в C есть собственные средства для определения адреса.

С помощью подпрограммы MPi_Type_get_contents можно определить фактические параметры, использованные при создании производного типа:

int MPI_Type_get_contents(MPI_Datatype datatype,

int max_integers, int max_addresses,

int max_datatypes, int *integers,

MPI_Aint *addresses, MPI_Datatype *datatypes)

MPI_TYPE_GET_CONTENTS(DATATYPE,

MAX_INTEGERS, MAX_ADDRESSES,

MAX_DATATYPES, INTEGERS,

ADDRESSES, DATATYPES, IERR)

Входные параметры:

- datatype — идентификатор типа;
- max_integers — количество элементов в массиве integers;
- max_addresses — количество элементов в массиве addresses;
- max_datatypes — количество элементов в массиве datatypes.

Выходные параметры:

- integers — содержит целочисленные аргументы, использованные при конструировании указанного типа;
- addresses — содержит аргументы address, использованные при конструировании указанного типа;
- datatypes — содержит аргументы datatype, использованные при конструировании указанного типа.

Подпрограмма MPI_Type_lb возвращает нижнюю границу типа данных datatype:

int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint Misplacement)

MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERR)

Выходным параметром является смещение (в байтах) нижней границы относительно источника displacement.

Верхнюю границу типа возвращает подпрограмма MPi_Type_ub:

int MPI_Type_ub(MPI_Datatype datatype,

MPI_Aint *displacement)

MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERR)

Еще один пример создания производного типа приведен в листинге 5.4.

Листинг 5.4. Пример создания производного типа в языке FORTRAN

PROGRAM MAIN_MPI

INCLUDE 'MPIF.H'

```
PARAMETER (N = 50)
REAL ARR(N, N, N) , B(9, 9, 9)
INTEGER SLICE1, SLICE2, SLICES, SIZEOFREAL
INTEGER RANK, IERR, STATUS(MPI_STATUS_SIZE)
INTEGER TAG, CNT, VCOUNT, BLOCKLEN, STRIDE, COUNT
CNT = 1
TAG = 0
VCOUNT = 9
BLOCKLEN = 1
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
IF (RANK.EQ.0) THEN
CALL MPI_TYPE_EXTENT(MPI_REAL, SIZEOFREAL, IERR)
STRIDE = 2
CALL MPI_TYPE_VECTOR(VCOUNT,
BLOCKLEN, STRIDE, MPI_REAL, SLICE1, IERR)
STRIDE = N * SIZEOFREAL
CALL MPI_TYPE_HVECTOR(VCOUNT,
BLOCKLEN, STRIDE, SLICE1, SLICE2, IERR)
STRIDE = N * N * SIZEOFREAL
CALL MPI_TYPE_HVECTOR(VCOUNT,
BLOCKLEN, STRIDE, SLICE2, SLICE3, IERR)
CALL MPI_TYPE_COMMIT(SLICES, IERR)
CALL MPI_SEND(ARR(1, 3, 2), CNT,
```

```
SLICES, 1, TAG, MPI_COMM_WORLD, IERR)

ELSE IF (RANK.EQ.I) THEN

COUNT = VCOUNT * VCOUNT * VCOUNT

CALL MPI_RECV(B, COUNT, MPI_REAL, 0,

TAG, MPI_COMM_WORLD, STATUS, IERR)

PRINT *, B

END IF

CALL MPI_FINALIZE(IERR)

STOP

END
```

В этой программе строятся производные типы slicel, slice2 и slice3, соответствующие сечениям исходного трехмерного массива А. Каким? Попробуйте самостоятельно определить производные структуры данных. Затем

между процессами в сечении выполняется блокирующий двухточечный обмен сообщениями.

Пример создания производного (структурного) типа приводится в листинге 5.5.

Листинг 5.5. Пример создания производного типа в языке С

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

struct newtype {

float a;

float b;

int n;

}

int main (int argc,char *argv[]) {

int myrank;
```

```
MPI_Datatype NEW_MESSAGE_TYPE;
int block lengths [3] ;
MPI_Aint displacements [3] ;
MPI_Aint addresses [4] ;
MPI_Datatype typelist [3] ;
int blocks_number;
struct newtype indata;
int tag = 0;
MPI_Status status;
MPI_Init (&argc, &argv) ;
MPI_Comm_rank (MPI_COMM_WORLD, smyrank) ;
typelist [0] = MPI_FLOAT;
typelist [1] = MPI_FLOAT;
typelist [2] = MPI_INT;
block_lengths[0] = block_lengths [1] = block_lengths [2] = 1;
MPI_Address (Sindata, ^addresses [0] ) ;
MPI_Address (& (indata. a) , saddresses [1] ) ;
MPI_Address (& (indata. b) , Saddresses [2] ) ;
MPI_Address (& (indata. n) , Saddresses [3] ) ;
displacements [0] = addresses [1] — addresses [0] ;
displacements [1] = addresses [2] — addresses [0] ;
displacements [2] = addresses [3] — addresses [0] ;
blocks number = 3;
MPI_Type_struct (blocks_number, block_lengths,
```

```
displacements, typelist, &NEW_MESSAGE_TYPE) ;

MPI_Type_commit (&NEW_MESSAGE_TYPE) ;

if (myrank == 0) {

indata.a = 3.14159; indata.b = 2.71828; indata.n = 2002;

MPI_Send(&indata, 1,NEW_MESSAGE_TYPE, 1, tag,

MPI_COMM_WORLD) ;

printf ("Process %i send: %f %f %i\n", myrank, indata.a, indata.b, indata.n) ;

}

else

{

MPI_Recv(&indata, 1, NEW_MESSAGE_TYPE, 0, tag,

MPI_COMM_WORLD, &status) ;

printf ("Process %i received: %f %f %i, status %s\n",

myrank, indata.a, indata.b, indata.n, status. MPI_ERROR) ;

}

MPI_Type_free (&NEW_MESSAGE_TYPE) ;

MPI_Finalize() ; return 0 ;

}
```

Здесь сначала задаются типы членов производного типа, а затем количество элементов каждого типа. После этого вычисляются адреса членов типа indata и определяются смещения трех членов производного типа относительно адреса первого, для которого смещение равно 0. Располагая этой информацией, можно определить производный тип, что и делается с помощью подпрограмм MPi_Type_struct и Mpi_Type_commit. Созданный таким образом производный тип можно применять в любых операциях обмена.

Создание и использование структурного типа в языке FORTRAN иллюстрируется программой, приведенной в листинге 5.6.

Листинг 5.6. Использование производного (структурного) типа в языке FORTRAN

```
PROGRAM MAIN_MPI

INCLUDE 'MPIF.H'

INTEGER RANK, TAG, CNT, IERR, STATUS (MPI_STATUS_SIZE)

PARAMETER (BUFSIZE=100)

CHARACTER RCVBUF1(BUFSIZE)

INTEGER NEWTYPE

INTEGER BLOCKS, POSITION

INTEGER DISP(2) , BLEN(2), TYPE(2)

INTEGER ADDRESS(2) INTEGER DATA1,

DATA3 COMPLEX DATA2, DATA4

CNT = 1

TAG = 0

CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)

IF(RANK.EQ.0) THEN

BLEN(I) = 1

BLEN(2) = 1

TYPE(I) = MPI_INTEGER

TYPE(2) = MPT_COMPLEX

BLOCKS = 2

CALL MPI_ADDRESS(DATA1, ADDRESS(1), IERR)

DISP(I) = ADDRESS(1)
```

```
CALL MPI_ADDRESS(DATA2, ADDRESS(2), IERR)
DISP(2) = ADDRESS(2)
CALL MPI_TYPE_STRUCT(BLOCKS,
BLEN, DISP, TYPE, NEWTYPE, IERR) CALL
MPI_TYPE_COMMIT(NEWTYPE, IERR)
DATA1 = 3
DATA2 = (1. , 3.)
CALL MPI_SEND(MPI_BOTTOM, CNT,
NEWTYPE, 1, TAG, MPI_COMM_WORLD, IERR)
PRINT *, "PROCESS", RANK,"SEND ", DATA1, " AND ", DATA2
CALL MPI_TYPE_FREE(NEWTYPE, IERR)
ELSE
POSITION = 0
CALL MPI_RECV(RCVBUF1, BUFSIZE,
MPI_PACKED, 0, TAG, MPI_COMM_WORLD,
STATUS, IERR)
CALL MPI_UNPACK(RCVBUF1, BUFSIZE,
POSITION, DATA3, 1, MPI_INTEGER,
MPI_COMM_WORLD, IERR)
CALL MPI_UNPACK(RCVBUF1, BUFSIZE,
POSITION, DATA4, 1, MPI_COMPLEX,
MPI_COMM_WORLD, IERR)
PRINT *, "PROCESS ", RANK, " RECEIVED ",
DATA3, " AND ", DATA4 END IF
```

```
CALL MPI_FINALIZE(IERR)

STOP

END
```

Результат выполнения этой программы выглядит так:

```
# mpirun -np 2 a.out PROCESS O SEND 3 AND (I.,3.)

PROCESS 1 RECEIVED 3 AND (I.,3.)
```

В профамме используется одна из подпрограмм упаковки и распаковки (MPIJJNPACK). К знакомству с этими подпрограммами мы и переходим.

Операции упаковки и распаковки данных

Альтернативным методом группировки данных по отношению к созданию производных типов можно считать использование подпрограмм упаковки и распаковки: MPI_Pack и MPI_unpack. Подпрограмма MPI_Pack позволяет явным образом хранить произвольные (в том числе и расположенные не в последовательных ячейках) данные в непрерывной области памяти (буфере передачи). Подпрограмму MPI_Unpack можно использовать для копирования данных из непрерывного буфера после их приема в произвольные (в том числе и не расположенные непрерывно) ячейки памяти.

Эти подпрограммы используются для обеспечения совместимости с другими библиотеками обмена сообщениями, для приема сообщений по частям, а также для того, чтобы буферизовать исходящие сообщения в пользовательское пространство памяти, что дает независимость от системной политики буферизации.

Следует учесть также, что сообщения передаются по коммуникационной сети, связывающей узлы вычислительной системы. Сеть эта работает довольно медленно, поэтому, чем меньше в параллельной профамме обменов, тем меньше потери на пересылку данных. С учетом этого желательно иметь механизм, который позволял бы, например, вместо отправки фех разных значений тремя сообщениями, отправлять их все вместе. Такие механизмы есть — в MPI их три. Это параметр count в подпрофаммах обмена, производные типы данных и подпрофаммы MPi_pack и MPi_unpack.

С помощью аргумента count в подпрофаммах MPi_send, MPi_Receive, MPi_Bcast и MPi_Reduce можно отправить в одном сообщении несколько однотипных элементов данных. Для этого элементы данных должны находиться в непрерывно расположенных ячейках памяти.

Ну, а если элементы данных — это простые переменные? Тогда они могут и не

находиться в последовательных ячейках памяти. В этом случае можно использовать производные типы данных или упаковку.

Подпрограмма упаковки может вызываться несколько раз перед передачей сообщения, содержащего упакованные данные, а подпрограмма распаковки в этом случае также будет вызываться несколько раз после выполнения приема. Для извлечения каждой порции данных применяется новый вызов. При распаковке данных текущее положение указателя в буфере сохраняется.

Интерфейс подпрограммы MPI_Pack выглядит следующим образом:

int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,

void *outbuf, int outcount, int *position, MPI_Comm comm)

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF,

OUTCOUNT, POSITION, COMM, IERR)

При вызове incount элементов указанного типа выбираются из входного буфера и упаковываются в выходном буфере, начиная с положения position. Входные параметры:

- inbuf — начальный адрес входного буфера;
- incount — количество входных данных;
- datatype — тип каждого входного элемента данных;
- outcount — размер выходного буфера в байтах;
- position — текущее положение в буфере в байтах;
- comm — коммуникатор для упакованного сообщения.

Выходным параметром является стартовый адрес выходного буфера outbuf.

Обратным действием обладает подпрограмма MPI_Unpack, которая используется для распаковки данных:

int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf,

int outcount, MPI_Datatype datatype, MPI_Comm comm)

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF,

OUTCOUNT, DATATYPE, COMM, IERR)

Входные параметры:

- inbuf — стартовый адрес входного буфера;
- insize — размер входного буфера в байтах;
- position — текущее положение в байтах;
- outcount — количество данных, которые должны быть распакованы;
- datatype — тип каждого выходного элемента данных;
- comm — коммуникатор для упаковываемого сообщения.

Выходной параметр outbuf — стартовый адрес выходного буфера.

Подпрограмма MPI_Pack_size позволяет определить объем памяти size (в байтах), необходимый для распаковки сообщения:

int MPI_Pack_size(int incount, MPI_Datatype datatype,

MPI_Comm comm,

int *size)

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERR)

Входные параметры:

- incount — аргумент count, использованный при упаковке;
- datatype — тип упакованных данных;
- comm — коммуникатор.

Пример использования подпрограмм упаковки приведен в листинге 5.7.

Листинг 5.7. Пример использования операций упаковки данных

```
#include "mpi.h"

#include <stdio.h>

int main(int argc,char *argv[])

{

int myrank;

float a, b;

int n;

int root = 0;
```

```
char buffer[100];

int position;

MPI_Init(Sargc, Sargv);

MPI_Comm_rank(MPI_COMM_WORLD, Smyrank);

if (myrank == root){ printf("Enter a, b, and n\n");

scanf("%f %f %i", &a, &b, sn); position = 0;

MPI_Pack(&a, 1, MPI_FLOAT, Sbuffer, 100, &position, MPI_COMM_WORLD);

MPI_Pack(&b, 1, MPI_FLOAT, Sbuffer, 100, &position, MPI_COMM_WORLD);

MPI_Pack(&n, 1, MPI_INT, &buffer, 100, &position, MPI_COMM_WORLD);

MPI_Bcast(&buffer, 100, MPI_PACKED, root, MPI_COMM_WORLD); }

else {

MPI_Bcast(Sbuffer, 100, MPI_PACKED, root, MPI_COMM_WORLD);

position = 0;

MPI_Unpack(&buffer, 100, &position, &a, 1, MPI_FLOAT, MPI_COMM_WORLD)

MPI_Unpack(&buffer, 100, &position, &b, I, MPI_FLOAT, MPI_COMM_WORLD)

MPI_Unpack(&buffer, 100, &position, &n, 1, MPI_INT, MPI_COMM_WORLD);

printf("Process %i received a=%f, b=%f, n=%i\n", myrank, a, b, n); }

MPI_Finalize(); return 0;
```

Здесь процесс 0 копирует в буфер сначала значение a, а потом дописывает туда b и n. После выполнения широковещательной рассылки главным процессом, остальные используют подпрограмму MPI_Unpack для извлечения *a, b* и я из буфера. Тип данных, который надо использовать при вызове

MPI_Bcast — MPI_PACKED.

Типы *MPI_BYTE* и *MPI_PACKED*

Особый тип данных MPI_BYTE не имеет аналога в C или FORTRAN. Он позволяет

хранить данные в "сыром" формате, имеющем одинаковое двоичное представление на стороне источника сообщения и на стороне адресата. Этот тип можно использовать при передаче любых сообщений в однородных системах.

Типы MPI_BYTE и MPI_PACKED не имеют соответствия в FORTRAN и C. Значение типа MPI_BYTE содержит байт, который в данном случае интерпретируется иначе, чем символ. Представление символа может различаться на разных машинах, а байт везде одинаков. Тип MPI_BYTE может использоваться, если необходимо выполнить преобразование между разными представлениями в гетерогенной вычислительной системе.

Атрибуты

Кэширование позволяет добавить к коммуникатору дополнительную информацию, которую называют *атрибутами* коммуникатора. Это дает возможность пересылать информацию, связывая ее с коммуникаторами любого типа (интра- и интер-). Коммуникатор, снабженный атрибутами, содержит группу процессов, контекст обмена и кэшированные данные (в том числе, возможно, и виртуальную топологию).

Если какая-то информация записана в качестве атрибута в одном из процессов, она автоматически становится доступной всем другим процессам из данного коммуникатора. Специальной организации пересылок сообщений при этом не требуется, система сама позаботится о передаче необходимой информации.

Атрибуты определяются только для того коммуникатора, к которому они присоединены. Они могут дублироваться только при дублировании коммуникатора, других способов их передачи от одного коммуникатора другому нет.

В языке C атрибуты имеют тип void *. Обычно это указатели на структуры, которые содержат необходимую информацию или обработчики для объекта MPI. В программах на языке FORTRAN атрибуты имеют тип INTEGER.

Идентификатором атрибута является целое число, которое назначается системой автоматически и называется *ключом атрибута.* Атрибуты бывают *системные* и *пользовательские.* Системные атрибуты не могут модифицироваться программистом, во всяком случае, напрямую. К их числу относятся топология, адрес обработчика ошибок и некоторые другие. Значения ключей являются глобальными и могут использоваться всеми коммуникаторами программы.

В MPI реализованы локальные операции с атрибутами. Подпрограмма

MPI_Keyval_create:

int MPI_Keyval_create(MPI_Copy_function *copy_fn,

MPI_Delete_function *delete_fn, int *keyval, void *extra_state)

MPI_KEYVAL_CREATE(COPY_FN,

DELETE_FN, KEYVAL, EXTRA_STATE, IERR)

предназначена для создания нового ключа атрибута keyvai (это выходной параметр). Ключи уникальны для каждого процесса и не видны пользователю, хотя явным образом хранятся в виде целых значений. Будучи однажды задан, ключ может быть использован для задания атрибутов и доступа к ним в любом коммуникаторе. Функция copy_fn вызывается, когда коммуникатор дублируется подпрограммой MPI_Comm_dup, а функция deiete_fn используется для удаления. Параметр extra_state задает дополнительную информацию (состояние) для функций копирования и удаления.

Если MPI-программа написана на двух языках (C и FORTRAN), функция copy_fn должна быть написана на том же языке, что и блок, из которого вызывалась подпрограмма MPI_Keyval_create.

Тип функции MPi_Copy_function определяется следующим образом:

typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyvai,

void *extra_state, void *attribute_val_in, void *attribute_val_out,

int *flag)

SUBROUTINE COPY_FUNCTION(OLDCOMM,

KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,

ATTRIBUTE_VAL_OUT, FLAG, IERR)

Функция копирования вызывается для каждого значения ключа в исходном коммуникаторе в произвольном порядке. Каждое обращение к функции копирования выполняется со значением ключа и соответствующим ему атрибутом. Если она возвращает значение флага flag = o, атрибут удаляется из продублированного коммуникатора. В противном случае (flag = 1) устанавливается новое значение атрибута, равное значению, возвращенному в параметре attribute_val_out.

Функцию copy_fn в языках C или FORTRAN можно определить значениями

MPI_NULL_COPY_FN ИЛИ MPI_DUP_FN. Значение MPI_NULL_COPY_FN является функцией, которая не выполняет никаких действий, только возвращает значение флага flag = o и MPI_SUCCESS. Значение MPI_DUP_FN представляет собой

простейшую функцию дублирования. Она возвращает значение флага flag = *i,* значение атрибута в переменной attribute_vai_out и код завершения

MPI SUCCESS.

Аналогичной copy_fn является функция удаления delete_fn. Она вызывается, когда коммуникатор удаляется вызовом MPI_Comm_free или при вызове MPI_Attr_delete. Эта функция должна иметь тип MPI_Delete_function, который определяется следующим образом:

typedef int MPI_Delete_function(MPI_Comm comm, int keyval,

void *attribute_val, void *extra_state);

SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL,

ATTRIBUTE VAL, EXTRA STATE, IERR)

Эта функция вызывается подпрограммами MPI_Comm_free, MPi_Attr_delete и MPI_Attr_put. Функция удаления может быть "пустой" — MPI_NULL_DELETE_FN. Функция MPI_NULL_DELETE_FN не выполняет никаких действий, возвращая только значение MPI_SUCCESS.

Специальное значение ключа MPI_KEYVAL_INVALID никогда не возвращается подпрограммой MPI_Keyval_create. Его можно использовать для инициализации ключей.

Удалить ключ keyval атрибута можно с помощью вызова подпрограммы

MPI_Keyval_free:

int MPI_Keyval_free(int *keyval) MPI_KEYVAL_FREE(KEYVAL, IERR)

Эта функция присваивает параметру keyval значение MPI_KEYVAL_INVALID. Используемый атрибут можно удалить, поскольку фактическое удаление происходит только после того, как будут удалены все ссылки на атрибут. Эти ссылки должны быть явным образом удалены программой, например, посредством вызова MPI_Attr_delete — каждый такой вызов удаляет один экземпляр атрибута, либо вызовом MPI_comm_free, который удаляет все экземпляры атрибута, связанные с удаляемым коммуникатором.

Подпрограмма MPI_Attr_put:

int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute)

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE, IERR)

используется для задания атрибута attribute, который в дальнейшем может применяться подпрограммой MPI_Attr_get. С атрибутом ассоциируется значение ключа keyval. Если значение атрибута уже задано, результат будет аналогичен ситуации, когда сначала для удаления предыдущего значения вызывается MPi_Attr_deiete (и выполняется функция обратного вызова delete_fn), а затем сохраняется новое значение. Вызов завершится с ошибкой, если нет ключа со значением keyval. В частности, MPI_KEYVAL_ INVALID — ошибочное значение ключа. Не допускается изменение системных атрибутов MPI_TAG_UB, MPI_HOST, MPI_IO и MPI_WTIME_IS_GLOBAL. Если ат-

рибут уже назначен, вызывается функция удаления, заданная при создании соответствующего ключа.

Подпрограмма MPi_Attr_get:

int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute, int *flag)

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE, FLAG, IERR)

возвращает значение атрибута attribute, соответствующее значению ключа keyval. Первый параметр задает коммуникатор, с которым связан атрибут. Если ключа со значением keyval нет, возникает ошибка. Ошибки не возникает, если значение key существует, но соответствующий атрибут не присоединен к коммуникатору comm. В этом случае возвращается значение флага

flag = false.

Вызов MPi_Attr_put передает в параметре attribute_val значение атрибута, а вызов подпрограммы MPi_Attr_get передает в параметре attribute_val адрес, по которому возвращается значение атрибута. Атрибуты должны извлекаться из программ, написанных на тех же языках, на которых они задавались с помощью вызова подпрограммы MPi_Attr_put.

Подпрограмма MPI_Attr_delete:

int MPI_Attr_delete(MPI_Comm comm, int keyval)

MPI_ATTR_DELETE(COMM, KEYVAL, IERR)

удаляет атрибут с указанным значением ключа. Делается это с помощью функции удаления атрибута deiete_fn, заданной при создании keyval. Параметр comm задает коммуникатор, с которым связан атрибут. Все аргументы данной подпрограммы входные. При любом дублировании коммуникатора с помощью подпрограммы

Mpi_comm_dup вызываются все функции копирования для атрибутов, установленных в данный момент времени. Делается это в произвольном порядке. Аналогичные действия выполняются при удалении коммуникатора вызовом MPi_Comm_free, но при этом вызываются все функции удаления.

Пример кэширования атрибутов приведен в листинге 5.8.

Листинг 5.8. Пример программы с кэшированием атрибутов

```
PROGRAM MAIN_MPI INCLUDE 'MPIF.H' INTEGER RANK,

IERR INTEGER NUMPROCS, NEWCOMM INTEGER KEY,

ATTR, EXTRA EXTERNAL COPY_FUN, DEL_FUN LOGICAL FLAG

CALL MPI_INIT(IERR)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)

CALL MPI_KEYVAL_CREATE(COPY_FUN, DEL_FUN, KEY, EXTRA, IERR)

ATTR = 120

CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEY, ATTR, IERR)

CALL MPI_ATTR_GET (MPI_COMM_WORLD, KEY, ATTR, FLAG, IERR)

PRINT *, "PROCESS = ", RANK, " BEFORE DUP ATTRIBUTE = ", ATTR

CALL MPI_COMM_DUP(MPI_COMM_WORLD, NEWCOMM, IERR)

CALL MPI_ATTR_GET(NEWCOMM, KEY, ATTR, FLAG, IERR)

IF (FLAG) THEN

PRINT *, "PROCESS = ", RANK, " AFTER DUP ATTRIBUTE = ", ATTR END IF

CALL MPI_COMM_FREE(NEWCOMM, IERR)

CALL MPI_FINALIZE(IERR)

STOP

END

SUBROUTINE COPY_FUN(COMM, KEYVAL,
```

```
EXTRA, ATTR_IN, ATTRJDUT, FLAG, IERR)
INTEGER COMM, KEYVAL, FUZZY, ATTR_IN, ATTRJDUT
LOGICAL FLAG
INCLUDE 'MPIF.H'
ATTRJDUT = ATTR_IN + 1
FLAG =.TRUE.
IERR = MPI_SUCCESS
RETURN
END
SUBROUTINE DEL_FUN(COMM, KEYVAL, ATTR, EXTRA, IERR)
INTEGER COMM, KEYVAL, ATTR, EXTRA, IERR
INCLUDE 'MPIF.H'
IERR = MPI_SUCCESS
IF(KEYVAL.NE.MPI_KEYVAL_INVALID) THEN
ATTR = ATTR - 1
END IF
RETURN
END
```

Результат выполнения этой программы:

```
#mpirun -np 4 a.out
PROCESS = 2 BEFORE DUP ATTRIBUTE = 120
PROCESS = 2 AFTER DUP ATTRIBUTE = 121
PROCESS = 1 BEFORE DUP ATTRIBUTE = 120
PROCESS = 1 AFTER DUP ATTRIBUTE = 121
```

PROCESS = 3 BEFORE DUP ATTRIBUTE = 120

PROCESS = 3 AFTER DUP ATTRIBUTE = 121

PROCESS = 0 BEFORE DUP ATTRIBUTE = 120

PROCESS = 0 AFTER DUP ATTRIBUTE = 121

Если использовать в качестве функций копирования и удаления MPI_

NULL_COPY_FN И MPI_NULL_DELETE_FN, результат изменится:

#mpirun -np 4 a.out

PROCESS = 2 BEFORE DUP ATTRIBUTE =5= 120

PROCESS = 1 BEFORE DUP ATTRIBUTE = 120

PROCESS = 3 BEFORE DUP ATTRIBUTE = 120

PROCESS = 0 BEFORE DUP ATTRIBUTE = 120

Попробуйте самостоятельно объяснить, почему изменился результат.

Ввод и вывод

В состав MPICH входит библиотека ROMIO, содержащая подпрограммы параллельного ввода/вывода и основные операции с файлами. Приведем краткий обзор некоторых подпрограмм из этой библиотеки. Используя их в своих программах, убедитесь, что библиотека ROMIO установлена на вашем компьютере.

Файл открывается с помощью подпрограммы Mpi_Fide_open для всех процессов из указанного коммуникатора:

int MPI_File_open(MPI_Corom comm, char *filename, int amode,

MPI_Info info, MPI_File *fh)

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERR)

Входные параметры:

- comm — коммуникатор;
- filename — имя файла;
- amode — режим доступа к файлу;
- info — информационный параметр.

Выходным является параметр fh — файловый дескриптор. Открытие файла является коллективной операцией, а коммуникатор обязательно должен быть интракоммуникатором. Во всех обращениях должны использоваться одинаковые значения параметров amode и filename, а значение info может быть разным.

Режимы доступа к файлу приведены в табл. 5.4.

Таблица 5.4. *Режимы доступа к файлу*

Режим	**Описание**
MPI_MODE_RDONLY	Открыть только для чтения
MPI MODE RDWR	Открыть для чтения и записи
MPI MODE WRONLY	Открыть только для записи
MPI MODE CREATE	Создать файл, если не существует
MPI_MODE_EXCL	Вернуть сообщение об ошибке, если создаваемый файл уже существует
MPI_MODE_DELETE_ON_CLOSE	Удалить файл после закрытия
MPI MODE UNIQUE OPEN	Открыть файл только один раз
MPI_MODE_SEQUENTIAL	Открыть файл только для последовательного

	доступа
MPI_MODE_APPEND	Установить указатель в конец файла

Каждый процесс "видит" файл в соответствии с тем представлением данных, которое используется на той платформе, на которой он выполняется. Преобразование форматов хранения данных не производится.

Информационный параметр используется для передачи дополнительной информации, относящейся к режимам ввода/вывода. Подробное описание зарезервированных значений можно найти в спецификации MPI-2. Здесь мы перечислим подпрограммы, с помощью которых создается информационный параметр, а в конце раздела приведем пример использования подпрограмм ввода/вывода.

Создается информационный параметр (info) с помощью подпрограммы

MPI_Info_create:

int MPI_Info_create(MPI_Info *info)

MPI_INFO_CREATE(INFO, IERR)

Этот параметр и является выходным.

Добавить пару (ключ, значение) в информационный параметр можно с помощью подпрограммы MPi_info_set:

int MPI_Info_set(MPI_Info info, char *key, char *value)

MPI_INFO_SET(INFO, KEY, VALUE, IERR)

Все параметры этой подпрограммы — входные.

Подпрограмма MPI_info_get_nkeys возвращает количество ключей, определенных в информационном параметре:

int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)

MPI_INFO_GET_NKEYS(INFO, NKEYS, IERR)

Входным параметром является информационный параметр info, а выходным — количество определенных ключей nkeys.

Получить значение n-го по счету ключа для информационного параметра

(info) можно с подпрограммы MPI_Info_get_nthkey:

int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)

MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERR)

Возвращается значение ключа (key).

Длина значения, соответствующего ключу, определяется с помощью подпрограммы MPI_Info_get_valuelen:

int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen,

int *flag)

MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERR)

Ее входные параметры:

- info — информационный параметр;
- key — ключ;

Возвращаются:

- valuelen — Длина аргумента value;
- flag — флаг, значение которого равно "истина", если ключ определен.

Извлечь значение (value), связанное с ключом, можно с помощью подпрограммы MPI_Info_get:

int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value,

int *flag)

MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERR)

Входные параметры:

- info — информационный параметр;
- key — ключ;
- valuelen —длина аргумента value.

Если ключ не определен, параметру flag присваивается значение "ложь".

Пара (ключ, значение) удаляется из информационного параметра подпрограммой MPI_Info_delete:

int MPI_Info_delete(MPI_Info info, char *key)

MPI_INFO_DELETE(INFO, KEY, IERR)

Ее входными параметрами являются:

- info — информационный параметр;
- key — ключ (строковое значение).

Подпрограмма MPI_Info_free уничтожает информационный параметр:

int MPI_Info_free (MPI__Info *info)

MPI_INFO_FREE(INFO, IERR)

Перейдем к другим подпрограммам для работы с файлами. Неблокирующее считывание из файла с использованием индивидуального файлового указателя выполняется с помощью подпрограммы MPi_Fiie_iread:

int MPI_File_iread(MPI_File fh, void *buf, int count,

MPI_Datatype datatype, MPIO_Request *request)

MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERR)

Ее входные параметры:

- fh — файловый дескриптор;
- count — количество элементов в буфере;
- datatype — тип каждого элемента в буфере.

Выходные параметры:

- buf — стартовый адрес буфера;
- request — идентификатор операции.

Неблокирующая запись выполняется подпрограммой MPI_File_iwrite:

int MPI_File_iwrite(MPI_File fh, void *buf, int count,

MPI_Datatype datatype, MPIO_Request *request)

MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERR)

Входные параметры:

- f h — файловый дескриптор;
- buf — стартовый адрес буфера;
- count — количество элементов в буфере;
- datatype — тип каждого элемента в буфере.

Выходным параметром является идентификатор операции request.

Закрыть файл fh можно с помощью подпрограммы MPi_Fiie_dose:

int MPI_File_close(MPI_File *fh) MPI FILE CLOSE(FH, IERR)

Подпрограмма MPi_File_deiete используется для удаления файла:

int MPI_File_delete(char *filename, MPI_Info info)

MPI_FILE_DELETE(FILENAME, INFO, IERR)

Ее входные параметры:

- filename — ИМЯ файла;
- info — информационный параметр.

Чтение из файла в блокирующем режиме с использованием индивидуального файлового указателя выполняется с помощью подпрограммы MPI_ File__read:

int MPI_File_read(MPI_File fh, void *buf, int count,

MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERR)

Входные параметры:

- fh — файловый дескриптор;
- count — количество элементов в буфере;
- datatype — тип элементов в буфере.

Выходные параметры:

- buf — начальный адрес буфера;
- status — статус.

Блокирующая запись в файл выполняется с помощью подпрограммы

MPI_File_write:

int MPI_File_write(MPI_File fh, void *buf, int count,

MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERR)

В качестве входных параметров задаются:

- fh — файловый дескриптор;
- buf — начальный адрес буфера;
- count — количество элементов в буфере;
- datatype — тип элементов.

Выходной параметр — статус операции status.

В листинге 5.9 приведен исходный текст одной из тестовых программ, входящих в состав ROMIO. В этой программе используются подпрограммы библиотеки. Рекомендуем читателю самостоятельно разобрать работу примера и попробовать выполнить эту программу.

Листинг 5.9. Исходный текст программы file_info.c

```
#include "mpi . h"

#include "mpio.h" /*эта строка не обязательна в версиях MPICH 1.1.1*/

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

/* Эта программа выводит значения всех ключей информационного параметра,

используемые по умолчанию в ROMIO */

int main(int argc, char **argv)

{

int i, len, nkeys, flag, mynod, default_striping_factor=0, nprocs;

MPI_File fh;
```

```
MPI_Info info, info_used;

char *filename, key [MPI_MAX_INFO_KEY] , value [MPI_MAX_INFO_VAL] ;

MPI_Init (Sargc, Sargv) ;

MPI_Comm_rank(MPI_COMM_WORLD, Smynod) ;

MPI_Comm_size (MPI_COMM_WORLD, Snprocs ) ;

/* Процесс с рангом 0 получает имя файла в качестве аргумента командной строки
запуска программы и передает его всем остальным процессам широковещательной
рассылкой */

if (Imynod) {

i = 1;

while { (i < argc) && strcmp("-fname", *argv) ) {

argv++; }

if (i >= argc) {

fprintf (stderr, "\n*# Usage: file_info -fname filename \n\n")

MPI_Abort (MPI_COMM_WORLD, 1) ; }

argv++;

len = strlen (*argv) ;

filename = (char *) malloc (len+1) ;

strcpy (filename, *argv) ;

MPI_Bcast(&len, 1, MPI_INT, 0, MPI_COMM_WORLD) ;

MPI_Bcast (filename, len+1, MPI_CHAR, 0, MPI_COMM_WORLD) ; }

else {

MPI_Bcast (&len, 1, MPI_INT, 0, MPI_COMM_WORLD) ;

filename = (char *) malloc (len+1) ;

MPI_Bcast( filename, len+1, MPI_CHAR, 0, MPI_COMM_WORLD) ;
```

```
/* Файл открывается с MPI_INFO_NULL */

MPI_File_open(MPI_COMM_WORLD, filename,

MPI_MODE_CREATE | MPI_MODE_RDWR,

MPI_INFO_NULL, &fh);

/* Проверка значений по умолчанию, установленных ROMIO */

MPI_File_get_info(fh, &info_used);

MPI_Info_get_nkeys(info_used, &nkeys);

for (i=0; Knkeys; i++) {

MPI_Info_get_nthkey(info_used, i, key);

MPI_Info_get(info_used, key, MPI_MAX_INFO_VAL-1, value, Sflag);

if (Imynod)

fprintf(stderr, "Process %d, Default: key = %s, value = %s\n", mynod, key, value);

if (!strcmp("striping_factor", key))

default_striping_factor = atoi(value); }

MPI_File_close(&fh); /* Удаление файла */

if (Imynod) MPI_File_delete(filename, MPI_INFO_NULL);

MPI_Barrier(MPI_COMM_WORLD);

/* Установка новых значений информационного параметра */

MPI_Info_create(&info);

/* Следующие четыре "подсказки" имеют смысл на всех машинах. Они могут быть заданы во время открытия файла или позже (многократно) */ /* Размер буфера для коллективного ввода/вывода */

MPI_Info_set(info, "cb_buffer_size", "8388608");

/* Количество процессов, которые действительно выполняют ввод/вывод в коллективной операции ввода/вывода */
```

```
sprintf(value, "%d", nprocs/2);

MPI_Info_set(info, "cb_nodes", value);

/* Размер буфера в индивидуальных операциях чтения */

MPI_Info_set(info, "ind_rd_buffer_size", "2097152"); /*

Размер буфера в индивидуальных операциях записи */

MPI_Info_set(info, "ind_wr_buffer_size", "1048576");

/* Следующие три подсказки относятся к расщеплению файла и имеют значение
только для файловых систем Intel PFS и IBM PIOFS, а в остальных случаях
игнорируются. Они могут быть заданы только в момент создания файла */ /* number of
I/O devices across which the file will be striped, accepted only if 0 < value <
default_striping_factor; ignored otherwise */

sprintf(value, "%d", default_striping_factor-l);

MPI_Info_set(info, "striping_factor", value); /* Размер блока в байтах */

MPI_Info_set(info, "striping_unit", "131072");

/* Номер устройства ввода/вывода, с которого начинается расщепление файла.
Принимается, только если 0 <= value < default_striping_factor, в противном случае
игнорируется */

sprintf(value, "%d", default_striping_factor-2);

MPI_Info_set(info, "start_iodevice", value);

/* Следующая подсказка имеет значение только для файловой системы Intel PFS и
сервера PFS. Она может быть задана в любой момент времени */

MPI_Info_set(info, "pfs_svr_buf", "true");

/* Открыть файл и задать новый информационный параметр */

MPI_File_open(MPI_COMM_WORLD, filename,

MPI_MODE_CREATE | MPI_MODE_RDWR, info, sfh);

/* Проверить значения */

MPI_File_get_info(fh, &info_used);
```

```
MPI_Info_get nkeys(info_used, snkeys);

if (Imynod) fprintf(stderr, "\n New values\n\n");

for (i=0; Knkeys; i++) { MPI_Info_get_nthkey(info_used, i, key);

MPI_Info_get(info_used, key, MPI_MAX_INFO_VAL-1, value, Sflag);

if (Imynod) fprintf(stderr, "Process %d, key = %s, value = %s\n", mynod, key, value);

}

MPI_File_close(&fh);

free(filename);

MPI_Info_free(&info_used);

MPI_Info_free(&info);

MPI_Finalize();

return 0;
```

Командная строка запуска этой программы выглядит следующим образом:

mpirun -np 2 a.out -fname aaa Здесь aaa — имя файла.

Вопросы и задания для самостоятельной работы

1. Проведите исследование быстродействия глобальных операций MPI для разного числа процессов и разных размеров сообщений.

2. Напишите программы, в которых коллективные операции обмена реализованы с помощью подпрограмм двухточечного обмена. Оцените трудоемкость программирования и эффективность программ. Напишите те же программы с использованием подпрограмм коллективного обмена. Сравните трудоемкость программирования и эффективность программ в обоих случаях.

3. Дан массив. Разработайте алгоритм и напишите MPI-программу вычисления максимального (минимального) элемента, не используя операции приведения MPI. То же самое сделайте с использованием операций приведения.

В следующих упражнениях требуется разработать алгоритм вычисления и написать MPI-программу, выбрав наиболее подходящие способы обмена. В упражнениях используются операции с матрицами и векторами. Читатель, не знакомый с высшей

математикой, может считать, что вектор — это одномерный массив, а матрица — массив двумерный. Действие матрицы *A* на вектор *b* определяется следующим образом:

C = Ab

где *c* — вектор, компоненты которого вычисляются по формуле:

$$c_i = \sum_{j=1}^{n} a_{ij} b_j,$$

здесь a_{ij}— элементы матрицы, которую считаем квадратной *(nxn).* Результатом матричного произведения является матрица:

C = AB, элементы которой вычисляются по формуле:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Скалярным произведением двух векторов a и *b* называют число:

$$p \equiv (a,b) = \sum_{i=1}^{n} a_i b_i.$$

4. Разработайте алгоритм и напишите MPI-программу вычисления скалярного произведения векторов *a* и b*.*

5. Разработайте алгоритм и напишите MPI-программу вычисления матричного произведения.

6. Даны матрицы *A и B.* Разработайте алгоритм и напишите MPI-программу вычисления матрицы:

C = *AB -BA.*

7. Дана матрица *A* и векторы *a* и *b.* Разработайте алгоритм и напишите MPI-программу вычисления:

P = (a, Ab).

8. Дана матрица *A* и векторы *a* и *b.* Разработайте алгоритм и напишите MPI-программу вычисления

c = a-Ab.

9. Имеется файл, содержащий записи для каждого работника. Каждая запись включает фамилию, имя, год рождения и год приема на работу. Разработайте алгоритм и напишите MPI-программу, в которой один из процессов распределяет всем остальным процессам приблизительно одинаковые "порции" информации, а эти процессы формируют список сотрудников, стаж которых составляет более 5 лет. Результаты пересылаются главному процессу, который их выводит в файл.

10. Необходимо смоделировать одномерное движение частиц в определенной области. Каждая частица характеризуется двумя значениями — координатой и скоростью. В начальный момент времени информация обо всех частицах находится на одном процессе, а затем распределяется поровну другим процессам, которые моделируют движение своей группы частиц. Напишите те части программы, которые выполняют передачу и прием информации о частицах.

ГЛАВА 6.

Введение в параллельное программирование с использованием PVM

Система разработки и выполнения параллельных программ PVM (Parallel Virtual Machine) является почти такой же распространенной системой, что и MPI, лишь немного уступая ей в популярности. "Родителями" PVM можно считать профессора университета Эмори (г. Атланта, США) Вэйди Санде-рама (Vaidy Sunderam) и сотрудника Окриджской Национальной Лаборатории Эла Гейста (Al Geist), а годом начала работ 1989-й. Затем к разработке системы подключились другие специалисты, в том числе и знаменитый Джек Донгарра (Jack Dongarra), человек, сделавший большой вклад в развитие современных компьютерных технологий, в том числе и высокопроизводительных вычислений. В этой главе мы познакомим читателя с тем, как "устроена" и как работает PVM. Расскажем о реализации модели передачи

сообщений, структуре библиотек. Разберем работу с PVM-консолью. Здесь же приводится пример простой PVM-программы и описание основных подпрограмм библиотеки. Сравнение PVM и MPI поможет читателю разобраться, в каком случае следует использовать MPI, а когда оптимальным будет выбор PVM. В заключение, как обычно, предлагаются задания для самостоятельной работы.

Общая характеристика PVM

PVM позволяет объединить разнородный набор компьютеров, связанных сетью, в общий вычислительный ресурс, который называют Параллельной Виртуальной Машиной. Отсюда и название системы. Компьютеры могут быть многопроцессорными машинами любого типа: суперкомпьютерами, рабочими станциями и т. д., объединенными сетями любого вида (Ethernet, ATM и др.). Как обычно, поддерживаются языки C/C++ и FORTRAN, хотя в настоящее время имеются средства сопряжения библиотек PVM и с другими языками, такими как Perl, Java, язык программирования математического пакета Matlab и др.

Гетерогенные вычислительные системы и PVM

В вычислительных системах и, прежде всего, в кластерах рабочих станций, практически всегда приходится встречаться сразу с несколькими видами *разнородности* (гетерогенности). Принято выделять пять видов разнородности:

- разнородность по архитектурам вычислительных узлов;
- разнородность по формату представления данных на разных вычислительных узлах;
- разнородность по быстродействию вычислительных узлов (прежде всего, производительности центральных процессоров);
- разнородность по загрузке узлов;
- разнородность но загрузке сети.

Первые три типа гетерогенности являются статическими, они не изменяются, если не изменяется аппаратное обеспечение вычислительной системы (например, менее производительные процессоры заменяются более производительными). Разнородность по архитектуре и формату представления данных может быть охарактеризована качественно, а разнородность по быстродействию допускает и количественную характеристику. Последние два вида разнородности динамические, они изменяются со временем. При программировании для разнородного кластера приходится сталкиваться с дополнительными, иногда достаточно сложными проблемами, поскольку для разных архитектур могут оказаться оптимальными различные способы и модели программирования, а наличие в системе разных форматов хранения данных затрудняет обмен данными. Программы, выполняющиеся на разных вычислительных узлах, могут в этом случае "не достичь взаимопонимания". Если на компьютерах установлены разные операционные системы, параллельная

программа должна транслироваться на каждом компьютере отдельно, что связано с несовместимостью операционных систем на уровне исполняемых бинарных файлов.

Учет динамической составляющей гетерогенности более сложен. Количество процессов на каждом компьютере, входящем в состав кластера, а следовательно, его загрузка, может изменяться, и прогнозировать это изменение практически невозможно. В любой момент, например, на любом компьютере из кластера может открыть сеанс работы новый пользователь. Это приведет к увеличению загрузки данного компьютера, возможно, увеличится загрузка сети, снизится эффективное быстродействие соответствующего вычислительного узла. Процессы, входящие в состав параллельного приложения, будут выполняться на разных хостах (напомним, что "хостом" мы называем отдельный вычислительный узел) с неодинаковой скоростью, а эффективность выполнения программы может значительно снизиться.

Все разновидности гетерогенности встречаются в кластерных вычислительных системах. Казалось бы, удобнее использовать симметричные многопроцессорные системы, однако у кластеров есть ряд преимуществ. Перечислим некоторые из них:

- относительно низкая стоимость — для создания кластера можно использовать уже имеющееся оборудование или сравнительно недорогие персональные компьютеры. Кластер из 10 рабочих станций дешевле десяти-процессорного суперкомпьютера и иногда — намного;
- производительность параллельной программы, которая выполняется на кластере, можно увеличить, используя для решения каждой конкретной подзадачи наиболее подходящую для нее архитектуру, выбирая подходящий хост из числа доступных;
- работа кластера более устойчива, поскольку выход из строя даже нескольких вычислительных узлов не обязательно приводит к остановке всей системы;
- кластер может наращиваться и модифицироваться в процессе эксплуатации постепенно, посредством замены отдельных его частей, без длительной остановки на замену оборудования.

Система PVM изначально создавалась с ориентацией именно на гетерогенные в любом смысле системы. Следует заметить, что и MPI поддерживает программирование для гетерогенных вычислительных систем, однако эта поддержка ограничена (она включает только поддержку гетерогенности по форматам хранения данных и в небольшой степени по архитектуре). В основу PVM положена идея *виртуальной параллельной вычислительной машины,* которая скрывает от программиста реальную неоднородность кластера, предоставляя в его распоряжение единый (но виртуальный!) многопроцессорный вычислительный комплекс. Этот виртуальный комплекс используется для реального выполнения реальных программ. Система берет на себя доставку сообщений, преобразование данных, управление процессами. PVM представляет собой систему передачи сообщений, которая развивалась независимо от стандарта MPI, своим путем. Сходство и различие обеих

систем рассматривается в заключительном разделе данной главы.

Разработчики системы положили в ее основу следующие принципы:

- управление конфигурацией виртуального параллельного компьютера должно осуществляться любым пользователем системы. Пользователь может добавлять в конфигурацию необходимое количество физических хостов по выбору, исходя из потребностей задачи, доступности хостов и степени их загрузки. Конфигурация может изменяться "на лету" в процессе работы, что позволяет, например, обеспечить более устойчивую работу программы, поскольку вышедший из строя компьютер может быть без труда исключен из виртуальной машины;
- "прозрачный" доступ к аппаратуре прикладных программ, для которых виртуальная машина представляет собой набор одинаковых процессорных элементов. С другой стороны, при необходимости, конкретные хосты могут использоваться для решения специфических задач, если это позволяет их архитектура или установленное на них программное обеспечение;
- единицей параллелизма в PVM является задача (или, в нашей терминологии, подзадача), состояние которой изменяется в процессе работы программы. Задача в определенные промежутки времени может выполнять вычисления, затем переходить к обмену данными и т. д.;
- использование явной модели обмена сообщениями. Размер сообщения ограничен лишь объемом доступной памяти;
- поддержка всех типов неоднородности;
- поддержка симметричных многопроцессорных систем.

Система PVM представляет собой набор библиотек и утилит, предназначенных для разработки и отладки параллельных программ, а также управления конфигурацией виртуальной вычислительной машины.

Основными целями, к которым стремились разработчики PVM, являются устойчивая работа системы, масштабируемость, поддержка любых типов гетерогенности и переносимость. Этих целей удалось достичь. PVM не может восстановить работу приложения после аппаратного сбоя, но дает программисту возможность написать программу, устойчивую к аппаратным сбоям и системным авариям. Виртуальная машина допускает динамическое изменение конфигурации. Масштабируемость виртуальной машины позволяет включать в ее состав сотни хостов и выполнять в ее среде тысячи процессов. PVM позволяет соединять вместе компьютеры разных видов. Она работает с минимальной модификацией в любой операционной системе из класса UNIX, а также в системах, построенных на общих принципах, таких как многозадачность и поддержка сетевой работы. Программный интерфейс прост, но имеет достаточную функциональность. Пакет может быть установлен обычным пользователем, не имеющим привилегий суперпользователя.

Программирование в PVM

Программирование с использованием PVM происходит следующим образом. Программист создает программу, которая при ее выполнении порождает несколько процессов, взаимодействующих между собой посредством обмена сообщениями. Как и в MPI, обмен и другие операции реализуются обращениями к подпрограммам библиотек PVM. Подпрограммы позволяют передавать и принимать сообщения, причем обмен ориентирован, вообще говоря, на выполнение программы в гетерогенной конфигурации. Данные, в общем случае разнотипные, перед передачей "упаковываются" в буфер передачи. При упаковке может использоваться кодирование данных. Напомним, что в MPI сообщение состоит из однотипных данных, а обмен между процессами, использующими для хранения данных разный формат, требует дополнительных усилий (тоже связанных с упаковкой данных).

В PVM реализованы уже знакомые читателю основные типы обмена -двухточечный и коллективный. Есть средства синхронизации процессов. В состав PVM, кроме того, входят подпрограммы управления конфигурацией виртуальной машины. Они позволяют, при необходимости, удалять из виртуальной машины хосты или добавлять в нее новые. Любая задача может запускать новые процессы или завершать выполнение других процессов, выполнивших свою работу. Все это вместе дает возможность эффективно использовать ресурсы параллельной вычислительной системы и создавать устойчивые по отношению к аппаратным сбоям программы. Все эти особенности сделали PVM одним из стандартов разработки параллельных программ, наряду с MPI и High Performance FORTRAN.

Архитектура PVM

Система PVM состоит из двух основных компонентов. Первым является демон pvmd3 (или pvmd), который запускается на всех компьютерах, входящих в состав виртуальной машины. Демонами в ОС UNIX называют процессы, управляющие различными службами, в данном случае, выполнением PVM-программ. Эти процессы работают без связи с терминалом, находясь в состоянии "сна" до тех пор, пока какой-нибудь процесс не обратится к соответствующей службе операционной системы. Особенностью демона pvmd3 является то, что запускать его (и даже устанавливать на компьютере) могут обычные пользователи, работающие без привилегий суперпользователя. Обычно же демоны являются системными процессами, подчиняющимися только суперпользователю. Если пользователь собирается запустить параллельную PVM-программу, он должен сначала запустить демоны на тех хостах, на которых будет выполняться параллельное приложение. В этом и заключается создание виртуальной машины.

Различные пользователи могут создавать собственные виртуальные машины, которые могут включать в свой состав одинаковые хосты, т. е. перекрываться, но являются абсолютно "прозрачными" друг для друга. "Прозрачность" заключается в том, что

процессы, выполняющиеся на разных виртуальных машинах, не могут взаимодействовать между собой, а их взаимное влияние опосредовано конкуренцией за одни и те же ресурсы (например, процессорное время или память). Запуск PVM-программы может выполняться из командной строки, как это делается обычно в ОС UNIX или из PVM-консоли. Специальный загрузчик приложений, как в MPI, не требуется.

Вторым компонентом PVM является набор библиотек, реализующих уже упоминавшиеся нами операции. Выше уже говорилось о том, что PVM поддерживает программирование на языках C/C++ и FORTRAN — основных языках программирования вычислений. В параллельных программах, написанных на языках C и C++, используются функции из библиотеки libpvmS. В программах на языке FORTRAN подпрограммы являются процедурами, но они фактически обеспечивают интерфейс с функциями C, поэтому при сборке программ, написанных на языке FORTRAN, необходимо подключать не только библиотеку libfpvm3 (FORTRAN), но и libpvm3 (C). Имена функций C начинаются с префикса pvm_, а имена процедур FORTRAN -с PVMF. В отличие от MPI значением функции C в PVM не обязательно является код завершения. Модель программирования PVM включает как SPMD, так и MPMD-модели, причем обе они равноправны.

Как работает PVM

После создания виртуальной параллельной машины на каждом хосте, включенном в ее состав, запускается демон pvmd. Во время выполнения PVM-программы каждому процессу, входящему в ее состав, системой присваивается свой уникальный *идентификатор задачи* TID (Task IDentifier). Этот идентификатор играет примерно ту же роль, что и ранг в MPI — он используется для адресации сообщений. В отличие от MPI, ответственность за назначение TID в PVM возлагается на локальные демоны pvmd. Это позволяет избежать дублирования значений идентификаторов задач при ошибочном их задании программистом. Определить значение идентификатора можно с помощью соответствующих подпрограмм. Строго говоря, идентификатор задачи является не просто номером процесса, он содержит дополнительную информацию, но программист может об этом не думать, считая TID просто адресом процесса. Не следует путать идентификатор TID с идентификатором процесса PID (Process IDentifier) в ОС UNIX, это независимые атрибуты исполняемых заданий в разных системах идентификации.

Каждой машине, на которой работает PVM, сопоставляется имя архитектуры. Имеются стандартные имена, но могут быть добавлены и новые. Иногда компьютеры с несовместимыми форматами исполняемых бинарных файлов используют одинаковые двоичные форматы представления данных. В PVM в этом случае не выполняется преобразование форматов.

Любой процесс в PVM может передавать сообщения любому другому процессу.

Количество и размер таких сообщений ограничены лишь размером доступной памяти.

PVM позволяет в процессе выполнения параллельной программы создавать *динамические группы процессов.* Динамическими они называются потому, что состав группы может меняться со временем. Процессы могут покидать группу и подключаться к ней. Это позволяет организовать операции обмена, область действия которых ограничена подмножеством процессов. Каждому процессу, вошедшему в состав группы, присваивается дополнительный групповой идентификатор. Групповые идентификаторы являются целыми числами, значения которых изменяются последовательно от 0 до g — 1, где g — размер группы. Группы могут перекрываться. Групповые функции содержатся в отдельной библиотеке libgpvm3.

При запуске демона PVM в каталоге /tmp создается блокирующий файл с именем pvmd.UID, где UID — числовой идентификатор пользователя, запустившего демон. Наличие данного файла является сигналом системе о том, что демон pvmd уже запущен. Если файл существует, а демон не запущен, при добавлении хоста в виртуальную машину система выведет сообщение об ошибке запуска демона. В этом случае следует войти в PVM-консоль (с PVM-консолью мы познакомимся чуть позже) и подать команду halt или вручную удалить блокирующие файлы из каталога /tmp. В каталоге /tmp создается также файл pvml.UID, содержащий протокол работы системы. Он может быть полезен при выяснении причин сбоев системы.

Демон PVM

На каждом хосте виртуальной машины запускается один демон pvmd. Если на одном компьютере работают несколько пользователей PVM, для каждого из них запускается собственный демон, который работает независимо от других.

Демон pvmd играет роль маршрутизатора сообщений. Он также заботится об управлении процессами и определении особых ситуаций. Демон продолжает работать, даже если работа PVM-программы завершилась аварийно, что облегчает отладку PVM-программ.

Самый первый демон pvmd, запущенный вручную, называют *главным* (master), все остальные — *подчиненными* (slaves). При нормальной работе системы все они равноправны, но только главному демону дается право запускать новые подчиненные демоны. Запросы об изменении конфигурации виртуальной машины, сформированные на произвольном хосте, направляются главному демону.

Идентификатор задачи

PVM использует идентификатор задачи TID для адресации демонов pvmd, задач и групп задач в рамках виртуальной машины. Идентификатор TID состоит из четырех полей (рис. 6.1), его длина 32 бита.

Поля S, G и H являются глобальными, т. е. каждый демон pvmd интерпретирует их одинаково. Поле H содержит номер хоста, который задается относительно виртуальной машины. При запуске каждый демон pvmd конфигурируется с собственным уникальным номером хоста.

Рис. 6.1. *Структура идентификатора задачи PVM*

Максимальное количество хостов в виртуальной машине ограничено значением 4095. Соответствие между реальными хостами и их номерами занесено в специальную глобальную хост-таблицу. Нулевое значение номера хоста используется для ссылки на локальный демон или для теневого демона pvmd.

Бит S используется для адресации демонов pvmd в случае, когда полю H присвоено значение номера хоста, а поле L пусто. Создатели PVM объясняют наличие этого бита "историческими причинами".

Каждому демону pvmd дается право использовать поле L для своих целей, оно также применяется для ссылки на локальный демон. Длина поля L — 18 бит, что дает возможность одновременно запускать на каждом хосте до 2^{18} - 1 задач. Это поле состоит из двух частей (см. рис. 6.1). Бит G используется для формирования адресов групповой рассылки (GID). Программист не имеет прямого доступа к полям TID.

Модель передачи сообщений в PVM

Демоны PVM и задачи могут обмениваться сообщениями произвольной длины. При обменах между хостами с разными форматами представления данные преобразуются в формат XDR. Формат XDR ("внешнее представление данных", external Data Representation) представляет собой стандартный метод обмена данными между компьютерами с разной архитектурой. Сообщению в момент передачи присваивается числовой идентификатор-тег, который назначается программистом. Теги используются для различения сообщений.

Источник сообщения не ожидает от адресата подтверждения приема, продолжая свою работу после того, как сообщение "выброшено" в коммуникационную сеть и буфер передачи можно использовать вновь. Перед приемом сообщения буферизуются на стороне адресата. Память для буферов обмена выделяется динамически. Последовательность сообщений от одного источника одному и тому же адресату сохраняется.

Асинхронные уведомления об особых событиях

PVM поддерживает механизм уведомлений об особых ситуациях в системе (табл. 6.1). Формированием уведомлений занимаются демоны pvmd.

Таблица 6.1. *Типы уведомлений об особых событиях*

Тип уведомления	**Описание**
PvmTaskExit	Задача завершена нормально или произошел сбой при ее выполнении
PvmHostDelete	Хост удален из виртуальной машины или произошел аппаратный сбой
PvmHostAdd	В виртуальную машину добавлены новые хосты

Настройка PVM и трансляция PVM-программ

Для установки PVM не требуются привилегии и права суперпользователя. Обычный пользователь может установить пакет в собственном каталоге. После установки необходимо правильно "настроить" систему, иначе она окажется неработоспособной. Прежде всего, пользователь должен установить значения двух переменных окружения PVM_ROOT и PVM_ARCH. Значение первой — путь к каталогу, содержащему PVM (этот каталог по умолчанию называется pvm3), а значение второй — наименование архитектуры данного хоста. В табл. 6.2 приведены некоторые из допустимых значений переменной PVM_ARCH.

Таблица 6.2. *Допустимые значения переменной PVM_ARCH*

Значение переменной PVM ARCH	**Вычислительная система**

ALPHA	DEC Alpha
BSD386	Персональный компьютер под управлением ОС BSD UNIX
CM2	CM2 фирмы Thinking Machines
CMS	CMS фирмы Thinking Machines
CNVX и CNVXN	Convex, серия C
CRAY	CRAY-90, YMP, T3D
CRAY2	CRAY-2
CRAYSMP	CRAY S-MP
HP3OO	HP-9000, модель 300
HPPA	HP-9000 PA-RISC
I860	Intel IPSC/860
LINUX	Персональный компьютер под управлением ОС LINUX
NEXT	NeXT
PGON	Intel Paragon
PMAX	DECstation3100, 5100

RS6K	IBM/RS6000
RT	IBMRT
SGI	IRIS IRIX 4.x фирмы Silicon Graphics
SGI5	IRIS IRIX 5.x фирмы Silicon Graphics
SGIMP	Мультипроцессор SGI
SUNS	Sun3
SUN4	Sun 4, SPARCstation
SUN4SOL2	Sun 4, SPARCstation Solaris 2.x
SUNMP	Мультипроцессор SPARC
UVAX	DEC Micro VAX

PVM можно установить и для тех архитектур, которые не входят в стандартный список, но для этого требуется дополнительная работа.

Для присвоения значений переменным окружения в ОС UNIX используются команды, которые могут немного различаться для разных интерпретаторов команд. В оболочке bash, например, это делается так:

export PVM_ROOT=/usr/share/pvm3 # export PVM_ARCH=LINUX

В приведенном примере предполагается, что PVM установлена в каталоге /usr/share. Для того чтобы не выполнять эти команды каждый раз в начале сеанса, рекомендуем включить их, а также другие команды инициализации в файл, который считывается в начале сеанса (.bashrc для интерпретатора bash и .cshrc для интерпретатора csh).

В каталоге $PVM_ROOT/lib имеются файлы cshrc.stub и bashrc.stub, которые следует присоединить к соответствующему конфигурационному файлу,

дописав его "в хвост файла" (листинг 6.1). Для этого можно использовать утилиту конкатенации UNIX cat или текстовый редактор. Дополнительно следует изменить значение переменной PATH, добавив в нее путь к каталогу bin, содержащему исполняемые файлы PVM.

Листинг 6.1. Пример дополнения к файлу .bashrc

```
#.bashrc

export PVM_ROOT=/usr/share/pvm3

export PVM_ARCH=LINUX

#

# append this file to your.bashrc to set path according to machine

# type, you may wish to use this for your own programs

(edit the last # part to point to a different directory f.e. ~/bin/_$PVM_ARCH.

#

if [ -z $PVM_ROOT ]; then if [ -d ~/pvm3 ]; then export PVM_ROOT=~/pvm3 else

echo "Warning — PVM_ROOT not defined"

echo "To use PVM, define PVM_ROOT and rerun your.bashrc" fi fi

if [ -n $PVM_ROOT ]; then

# Определяется архитектура

export PVM_ARCH='$PVM_ROOT/lib/pvmgetarch` #

В путь поиска включаются каталоги, содержащие исполняемые файлы

export PATH=$PATH:$PVM_ROOT/lib

export PATH=$PATH:$PVM__ROOT/bin/$PVM_ARCH fi

# Следующая строка нужна, если используется
```

XPVM export XPVM_ROCT=/home/pub/pvm/xpvm

export PATH=$PATH:/usr/share/pvm3/lib/LINUX

Кроме переменных окружения PVM_ROOT и PVM_ARCH имеются и другие:

- PVM_EXPORT — имена переменных окружения, которые следует экспортировать из родительской задачи в дочерние, запущенные при вызове подпрограммы pvm_spawn. Имена в списке должны разделяться двоеточием. Если значение этой переменной не установлено, переменные не экспортируются;
- PVMJDEBUGGER — полное абсолютное имя отладочного командного файла, который необходимо использовать при вызове подпрограммы pvm_spawn с ключом PvmTaskDebug. По умолчанию $PVM_ROOT/lib/debugger;
- PVM_DPATH — содержит путь к стартовому командному файлу pvmd (по умолчанию $PVM_ROOT/lib/pvmd). Значение этой переменной может быть перекрыто ключом dx= в файле описания виртуальной машины (речь об этом файле пойдет ниже);
- PVMDLOGMAX — устанавливает максимальную длину файла протокола ошибок демона pvmd. По умолчанию 1 Мбайт;
- PVMDDEBUG — устанавливает маску отладки демона pvmd (но не прикладной программы!), используемую по умолчанию (аналогично ключу pvmd -d). Значение задается в шестнадцатеричном, восьмеричном или десятичном формате;
- PVMTASKDEBUG — устанавливает маску отладки библиотеки libpvm, используемую по умолчанию (аналогично вызову подпрограммы pvm_setopt (FvmDebugMask, x)). Значение задается в шестнадцатеричном, восьмеричном или десятичном формате;
- PVMBUFSIZE — размер буферов в разделяемой памяти, используемых библиотекой libpvm и демоном pvmd. По умолчанию используется значение 1 Мбайт;
- PVMKEY — используется PVM для служебных целей.

Следующие переменные не подлежат модификации (кроме PVM_ARCH):

- PVM_ARCH — имя архитектуры, для которой установлена PVM;
- PVMSOCK — передается от демона pvmd к запущенной задаче и содержит адрес локального сокета pvmd;
- PVMEPID — ожидаемый идентификатор процесса для запущенной задачи;
- PVMTMASK — маска трассировки libpvm, которая передается от pvmd к запущенным задачам.

Необходимо также создать файл .rhosts таким же образом, как и при настройке MPICH. Можно также добавить каталог $PVM_ROOT/man к переменной поиска справочных страниц MANPATH.

Трансляция PVM-программ выполняется следующим образом:

f77 -o dot dot.f -L/usr/local/pvmS/lib -IfpvmS -Ipvm3 для программ на языке FORTRAN и

gcc -o dot dot.f -L/usr/local/pvm3/lib -Ipvm3

для программ на языке С.

Во втором случае не требуется подключение библиотеки HbfpvmS, которая используется только в программах FORTRAN. Ключ -о позволяет задать явным образом имя исполняемого файла программы, ключ *-L<nymb к библиотеке>* указывает расположение библиотечных файлов, а ключ *-1<имя библиотеки>* определяет имя подключаемой библиотеки. Можно использовать и другие ключи, их описание можно найти в справочных руководствах по соответствующим трансляторам, но вышеупомянутые ключи надо использовать обязательно.

Структура каталога PVM

Файлы PVM могут быть помещены, например, в каталог /usr/local/pvm3 или в домашний каталог какого-нибудь пользователя. В" каталоге pvm3 обычно имеются следующие подкаталоги:

- pvm3/bin/$PVM_ARCH — исполняемые файлы (обычно это примеры из дистрибутива, установленные системным администратором);
- pvm3/conf — конфигурационные файлы для архитектур, поддерживаемых PVM;
- pvm3/console — исходные файлы PVM-консоли;
- pvm3/doc — документация;
- pvm3/examples — исходные тексты примеров PVM-программ;
- pvm3/gexamples — исходные тексты примеров программ, использующих групповые операции;
- pvm3/include — заголовочные файлы для PVM-программ;
- pvm3/lib — исполняемые файлы (в основном командные файлы);
- pvm3/lib/$PVM_ARCH — исполняемые файлы системы (pvmd, консоль и др.);
- pvm3/libfpvm - исходные тексты библиотеки libfpvm для языка FORTRAN;
- pvm3/man/man — справочные страницы;
- pvm3/pvmgs - исходные тексты библиотеки libgpvm и сервера имен группы;
- pvm3/src — исходные тексты библиотеки libpvm и демона pvmd.

Примеры, входящие в состав дистрибутива PVM, представляют собой исходные тексты программ, которые можно использовать как для изучения приемов программирования в PVM, так и в качестве "шаблонов" при разработке собственных программ. В любом случае рекомендуется скопировать файлы с примерами в свой домашний каталог:

I cp -r $PVM_ROOT/examples $HOME/pvm3/examples

cd $HOME/pvm3/examples

Для трансляции и сборки исполняемых файлов используется специальный командный файл aimk, выполняющий необходимые действия, сценарий которых описан в файле Makefile.aimk. Пример использования этой утилиты (она является аналогом утилиты make, хорошо знакомой тем программистам, кто работает в среде ОС UNIX):

I aimk fmaster fslave

Здесь транслируется параллельная программа (на языке FORTRAN) fmaster/fslave, построенная по схеме "главная/подчиненная задачи". В качестве аргументов утилиты aimk используются имена программ-примеров.

Консоль PVM

PVM-консоль — это программа, которая позволяет работать с виртуальной параллельной машиной в интерактивном режиме. Консоль можно многократно запускать и прерывать работу с ней с помощью команды quit на любом из хостов виртуальной машины. Ни конфигурация виртуальной машины, ни состояние выполняющихся PVM-программ при этом не изменятся.

Консоль при первом запуске загружает демон pvmd и в дальнейшем взаимодействует с ним. При включении в конфигурацию нового хоста, демон pvmd загружается на нем. Командная строка запуска PVM-консоли имеет вид:

pvm [-n <имя_хоста>] [hostfile]

Ключ -n используется для указания альтернативного имени главного демона pvmd, если имя хоста не соответствует требуемому IP-адресу.

Файлы $PVM_ROOT/lib/pvm и $pvM_ROOT/lib/pvmd являются командными файлами, которые используются для запуска консоли PVM и демона pvmd. Они определяют архитектуру компьютера и запускают уже настоящие программы из каталога $pvM_ROOT/lib/$pvM_ARCH.

При запуске консоли сначала выводится вспомогательная информация примерно такого вида:

#

example PVM console startup script

copy this file to $HOME/.pvmrc

```
#
# command aliases #
alias ? help
alias print_environment spawn -> /bin/env
alias h help
alias_ jobs
alias t ps
alias tm trace
alias v version
#
# important for debugging
#
setenv PVM_EXPORT DISPLAY
#
# want to see these trace.events by default
#
tm addhosts delhosts halt
tm pvm_mytid pvm_exit pvmjparent
tm send recv nrecv probe mcast trecv sendsig recvf
version # print PVM release version
3.4.3
id # print console TID
t40004
```

Затем, в случае успешного запуска консоли, появляется приглашение:

pvm>

После этого можно вводить команды, список которых приведен в табл. 6.3.

Таблица 6.3. *Команды интерактивного режима PVM-консоли*

Команда	**Описание**
add *хост1 . . .* alias *[псевдокоманда]*	Добавляет указанный хост (или хосты, если указано несколько имен) в виртуальную машину Задает или выводит (если аргументов нет) список псевдокоманд PVM (аналог псевдокоманд ОС UNIX)
conf	Выводит список хостов, входящих в виртуальную машину. Список содержит имя хоста, идентификатор задачи (ТЮ) демона pvmd, архитектуру и относительное быстродействие в "условных единицах"
delete *хост1 . . .*	Удаляет указанный хост (или хосты, если указано несколько имен) из виртуальной машины. Если на этих хостах все еще выполняются процессы PVM-программы, они будут завершены (подобных "насильственных" методов завершения работы параллельной программы следует избегать)
echo *аргумент*	Эхо-отображение аргументов команды (аналог команды echo в ОС UNIX)

halt	Завершает работу всех процессов PVM, в том числе консоли и всех демонов. После завершения работы параллельных программ виртуальная машина должна быть удалена с помощью именно этой команды
help *[команда]*	Выводит краткую (очень краткую!) справку по указанной команде или выводит список всех команд (если команда help вводится без аргументов)
id	Выводит идентификатор задачи для PVM-кон-соли
jobs	Выводит список задач
kill	Завершает работу процесса PVM
mstat *хост1. . .*	Выводит состояние хостов, имена которых указаны в качестве аргументов
ps -a	Выводит список процессов, выполняющихся в данный момент на виртуальной машине, включая имена хостов, на которых они выполняются, идентификаторы задач и идентификаторы родительских задач
pstat	Выводит информацию о состоянии одного процесса
	Завершает работу с PVM-консолью

quit	без удаления виртуальной машины и без завершения работы параллельных программ
reset	Завершает работу всех процессов PVM за исключением консоли с последующей инициализацией внутренних таблиц PVM и очередей сообщений. Демоны находятся в состоянии "сна"
setenv *[переменная= значение]*	Задает значения переменных окружения или выводит их значения (если аргументы отсутствуют)
sig [номер *сигнала]* [TID] spawn [*ключи]*	Передает сигнал процессу с заданным идентификатором Запускает прикладную PVM-программу. Допускается использование следующих ключей: • *-число*— количество запускаемых копий одной задачи, по умолчанию 1; • -хост— имя хоста, на котором следует запустить процесс. По умолчанию используется любой хост, входящий в виртуальную машину; • *-архитектура*— запустить задачу на хосте с указанной архитектурой; • - ? — включить режим отладки; • -> — перенаправить стандартный вывод задач на PVM-консоль; • -> *файл*— перенаправить стандартный вывод задач в файл, перезаписав его, если он

trace [маска] unalias [*псевдокоманда]* Version	уже существует; • -» *файл*— перенаправить стандартный вывод задач в файл, добавив новые записи в конец файла, если он уже существует; • -@ — трассировка программы с отображением вывода на PVM-консоли; • -@ *файл* — трассировка программы с записью результата в файл Устанавливает или выводит (если аргументы не заданы) маску трассировки событий Отменяет псевдокоманду Выводит версию PVM

Обычно сразу же, после первого в текущем сеансе работы с виртуальной машиной запуска консоли, создается виртуальная конфигурация. Сделать это можно с помощью команды add, в качестве аргумента которой указываются сетевые имена компьютеров, например:

pvm> add dOl d02 2 successful

HOST DTID

dOl 140000

d02 180000

В ответ система должна вывести сообщение об успешном добавлении хостов. Иногда при выполнении этих команд можно получить диагностическое сообщение "Can't Start pvmd". Это сообщение говорит о том, что возникли проблемы с запуском демона. Чуть позже мы приведем возможные причины этих проблем.

Удаление хоста из виртуальной машины выполняется командой delete с указанием его имени:

pvm> delete d01

С помощью команды conf можно получить информацию о конфигурации виртуальной

машины:

pvm> conf

4 hosts, 1 data format

HOST DTID ARCH SPEED DSIG

f08 40000 LINUX 1000 0x00408841

f01 80000 LINUX 1000 0x00408841

f02 c0000 LINUX 1000 0x00408841

f03 100000 LINUX 1000 0x00408841

Здесь в первом столбце выводится имя хоста, во втором — идентификатор демона DTID, в третьем — наименование архитектуры хоста, затем относительная скорость (в условных единицах) и в четвертом столбце — сигнатура данных (DSIG).

Запуск параллельной программы можно выполнить из консоли или из командной строки. В последнем случае необходимо выйти из консоли с помощью команды quit.

При запуске программы из консоли система ищет исполняемый файл в каталоге $HOME/pvm3/bin/$PVM_ARCH. Туда и следует помещать откомпилированные программы. Для запуска программы используется команда spawn. Если в каких-либо подзадачах параллельной программы предусмотрен вывод на терминал, необходимо использовать ключи -> или ->> для "перехвата" вывода. Пример использования команды spawn для запуска программы из листингов 6.5 и 6.6 приводится ниже:

pvm> spawn -> a.out [7:t80008] Master's tid 524296

[7:t80008] Recieved from slave with tid 524297: Hi,

PVM Programmer! [7:t80009] EOF [7:t80008] EOF

При отладке можно включить режим трассировки с записью результата в файл, поскольку на экран в этом случае выводится обычно большой объем информации, например:

pvm> spawn -8 file.trace a.out

В результате выполнения этой команды будет создан файл file.trace, фрагмент которого приводится в листинге 6.2.

Листинг 6.2. Фрагмент файла трассировки программы из листингов 6.S и 6.6

```
/*
* "Creation Date" "Fri Jun 28 10:53:23 2002"
* "Machine" "Console 3.4.3 (PVM 3.4.3)" */;;
#1802: "host_add(0)"
{
// "TS" "Time seconds"
int "TS";
// "TU" "Time Microseconds"
int "TU";
// "TID" "Task ID"
int "TID";
// "HN" "Host Name"
char "HN"[];
// "HNA" "Host Name Alias"
char "HNA"[];
// "HA" "Host Architecture"
char "HA"[];
// "HSP" "Host Speed"
int "HSP";
};;
"host_add(0)" { 1025261603, 481722, 262144, [4] { "f08" }, [4] { "f08" },
[6] { "LINUX" }, 1000 };;
"host_add(0)" { 1025261603, 481722, 524288, [4] { "fOI" }, [4] { "fOI" },
[6] { "LINUX" }, 1000 };;
```

#1806: "host_sync(0)"

#1150: "newtask(0)" {

// "TS" "Time Seconds"

int "TS";

// "TU" "Time Microseconds"

int "TU";

// "TID" "Task ID"

int "TID";

// "PT" "Parent Task ID"

int "PT";

// "TF" "Task Flags" int "TF";

// "TN" "Task Name" char "TN"[];

};;

"newtask(0)" { 1025261603, 484505, 262159, -35, 64, [6] { "a.out" } };;

#1808: "output(0)"

"spntask(0)" { 1025261603, 485322, 262159, -35 };;

"output(0)" { 1025261603, 485322, 262159, [9] { "GOTSPAWN" } };;

#1052: "mytid(0.0)"

"mytid(0.0)" { 1025261603, 499219, 262159 };;

"endtask(0)" { 1025261603, 523065, 262159, 0, 0, 0, 0, 0 };;

"endtask(0)" { 1025257916, 71554, 524298, 0, 0, 0, 0, 0 };;

При выходе из консоли по команде quit виртуальная машина остается, при этом на любом хосте, входящем в ее состав, можно вновь запустить консоль (система выведет сообщение "pvm already running") и продолжить работу в интерактивном режиме. Если же для выхода используется команда halt, виртуальная машина прекращает свое существование, а процессы, если они были, завершат свою работу.

Возможные проблемы при работе с PVM

Если при попытке добавления в виртуальную машину нового хоста выводится сообщение вида:

[t80020000] Can't start pvmd

оно говорит о невозможности запуска демона. В этом случае следует проверить, имеется ли в домашнем каталоге файл .rhosts и правильно ли он подготовлен, в том числе, правильно, ли заданы права доступа к этому файлу (их символьное представление должно иметь вид r----------, что означает право на чтение для владельца файла и отсутствие всех остальных прав доступа). Признак конца файла должен находиться в строке, следующей за последней строкой с именем хоста. Файл создан правильно, если при выполнении команды:

rsh d01 ls

где d01 — имя удаленного компьютера, на экран будет выведено содержимое домашнего каталога пользователя на этом компьютере. Причина может

быть и другой — неправильное значение переменной PVM_ROOT. Рекомендуется выполнить следующую проверку:

rsh d01 $PVM_ROOT/lib/pvmd

Это команда запуска демона pvmd на компьютере d01. Убедитесь также, что пакет PVM действительно установлен. Еще одна причина отказа в запуске демона — оставшийся при некорректном завершении работы с PVM (ручная остановка системы или системная авария) блокирующий файл в каталоге /tmp данного хоста.

При попытке запуска PVM может вернуть пользователю сообщение:

[t80020000] Login incorrect

В этом случае пользователь, скорее всего, не зарегистрирован на удаленном компьютере под указанным именем. Данная проблема может быть решена визитом к системному администратору.

Конфигурационный файл PVM-консоли

При запуске консоли программа считывает конфигурационный файл $HOME/.pvmrc. Этот файл может содержать определения псевдокоманд, переменных окружения, а также команд, которые будут выполняться при запуске консоли. Пример файла .pvmrc приведен в листинге 6.3.

Листинг 6.3. Пример конфигурационного файла PVM-консоли

```
#

# command aliases #

alias ? help

alias print_environment spawn -> /bin/env

alias h help

alias j jobs

alias t ps

alias tm trace

alias v version

#

# important for debugging #

setenv PVM_EXPORT DISPLAY #

# want to see these trace events by default #

tm addhosts delhosts halt

tin pvmjnytid pvm_exit pvm_parent

tm send recv nrecv probe mcast trecv sendsig recvf

version # print PVM release version

id # print console TID
```

Файл описания виртуальной машины

Каждый пользователь PVM. может создать собственный файл с описанием персональной виртуальной машины *(хост-файл,* в дальнейшем для краткости мы так и будем его называть). Это следует сделать, если часто приходится запускать программу на виртуальной машине одной и той же конфигурации. Хост-файл является текстовым, как и другие конфигурационные файлы операционной системы UNIX. В каждой строке на первом месте находится сетевое имя компьютера. На каждое имя

отводится по одной строке. Если строка начинается с символа #, она считается строкой комментария. Пустые строки игнорируются. Кроме имени хоста, в строке могут быть дополнительно указаны ключи. Их перечень дан в табл. 6.4.

Таблица 6.4. *Ключи, используемые в файле описания виртуальной машины (хост-файле)*

Ключ	**Описание**
lo= *регистрационное имя*	Использует при работе на данном хосте указанное регистрационное имя. По умолчанию применяется то же имя, что и на локальном хосте
so= pw	Запрашивать пароль при входе на указанный хост. Используется, если у пользователя на разных компьютерах разные регистрационные имена и пароли. По умолчанию, для запуска демонов pvmd на удаленных машинах PVM использует rsh. С данным ключом вместо rsh будет применяться rexec
dx= *положение демона pvmd*	Задает положение демона, отличное от используемого по умолчанию
ep= *путь к каталогу с исполняемыми файлами*	Позволяет задать пути поиска исполняемых файлов PVM-программ. Если задаются несколько путей, они разделяются двоеточиями. По умолчанию используется каталог $HOME/pvm3

	/bin/PVM_ARCH
sp= *число*	Относительное быстродействие хоч. га в данной конфигурации. Задается целочисленным значением от 1 до 1000000. По умолчанию 1000
bx= *положение отладчика*	Задает положение командного файла отладчика, который вызывается, если в подпрограмме запуска (pvm_spawn) используется ключ отладки. Положение отладчика может быть задано и с помощью переменной окружения PVM DEBUGGER. По умолчанию pvm3/lib/debugger
wd= *рабочий каталог*	Рабочий каталог для PVM-программ. По умолчанию $HOME
ip= *имя хоста* so= ms	Имя хоста, соответствующее его IP адресу Подчиненный демон должен запускаться на данном хосте вручную (manual start). Ручной старт применяется, если не работают rsh и гехес

В поле имени хоста может находиться звездочка. Она играет роль шаблона. В этом случае действие ключей распространяется на все хосты, имена которых указаны в следующих строках.

Если в начале строки находится символ &, соответствующий хост не включается в начальную конфигурацию виртуальной машины. Если он будет добавлен в конфигурацию впоследствии, будут использованы и заданные в строке для этого хоста ключи. Пример хост-файла приведен в листинге 6.4.

Листинг 6.4. Пример хост-файла

```
f01.ptc.spbu.ru

f02.ptc.spbu.ru

d01.icape.nw.ru dx=/usr/share/pvm3/lib/I860/pvmd3

d02.icape.nw.ru lo=romano so=pw

d03.icape.nw.ru lo=andrei so=pw

d04.icape.nw.ru lo=mariya so=pw

d05.icape.nw.ru lo=yuri so=pw

d06.icape.nw.ru lo=valeri so=pw

&sun4 ep=quarks
```

Наша первая PVM-программа

Технология создания параллельной PVM-программы аналогична технологии программирования с использованием MPI. Программист пишет одну или несколько последовательных программ с обращениями к подпрограммам библиотек PVM. Каждая программа транслируется на тех хостах, которые будут включены в состав виртуальной машины. Раздельная трансляция требуется, если различается архитектура хостов. Исполняемые файлы программы размещаются в определенных каталогах. Затем обычно запускается одна главная задача, которая производит запуск остальных, подчиненных задач. Возможны и другие варианты, такие, например, как запуск вручную всех процессов, относящихся к параллельной программе. Взаимодействие между процессами осуществляется посредством обмена сообщениями.

В листингах 6.5 и 6.6 приведен пример простой PVM-программы. Она работает следующим образом. После запуска на экран выводится идентификатор задачи, соответствующий запущенному процессу, затем уже из главной задачи производится запуск другого процесса (вызов подпрограммы pvm_spawn). В случае успешного выполнения подпрограммы pvm_spawn, главная задача принимает сообщение, которое представляет собой строку символов с приветствием "Hi, PVM Programmer!". Подчиненная программа должна быть откомпилирована так, чтобы имя ее исполняемого файла было greeting.

Листинг 6.5. Пример PVM-программы

```
# include "pvm3.h"
main ( )
{
int ierr, tid, msgtag;
char buf [100] ;
printf ("Master's tid %i\n", pvm_mytid ( ) ) ;
ierr = pvm_spawn( "greeting", (char**)0, 0, "", 1, stid) ;
if (ierr == 1) {
msgtag = 1;
pvm_recv(tid, msgtag);
pvm_upkstr (buf ) ;
printf ("Recieved from slave with tid %i: %s\n", tid, buf);
}
else
printf ("Can't start hello_other\n") ;
pvm_exit ( ) ;
}
```

Листинг 6.6. Пример PVM-программы. Подчиненная задача greeting

```
# include "pvm3.h"
main()
{
int ptid, msgtag;
char buf[100];
ptid = pvm_parent();
```

```
strcpy(buf, "Hi, PVM Programmer!");

msgtag = 1;

pvm_init5end(PvmDataDefault);

pvm_pkstr(buf);

pvm_send(ptid, msgtag);

pvm_exit();
```

В начале PVM -программы должен находиться оператор include, который подключает заголовочный файл с необходимыми описаниями так же, как это делается в MPI-программах. В программах на языке FORTRAN должен быть указан полный (абсолютный) путь к заголовочному файлу:

$PVM_ROOT/include/fpvm3 . h

Краткая форма допускается, только если выполнено условие:

$PVM_ROOT = $HOME/pvm3

Напомним, что исполняемые файлы программы должны быть помещены в каталог:

$HOME/pvm3 /bin/ PVM_ARCH

Этот каталог по умолчанию используется системой PVM для размещения исполняемых программ. Если работа происходит в ОС Linux, имя самого внутреннего каталога — LINUX (все буквы имени в верхнем регистре).

Для того чтобы разобраться с работой этой программы, познакомимся с основными подпрограммами PVM.

Управление процессами

Определить идентификатор задачи для данного процесса можно с помощью подпрограммы pvm_mytid:

int tid = pvm_mytid(void) PVMFMYTID(TID)

Обычно вызов данной подпрограммы является первым обращением к подпрограммам PVM, в этом случае он не только определяет идентификатор задачи, но и подключает программу к PVM. При подключении генерируется значение TID. Вообще говоря, это происходит при вызове любой подпрограммы PVM, если он первый, однако чаще всего вызывается в начале программы именно pvm_mytid, что требуется для

определения процессом своей роли в коллективе процессов параллельной программы. Отрицательное значение TID возвращается при возникновении ошибки. Вместо численного значения кода удобнее использовать именованные константы. Перечень некоторых ошибок приводится в конце этого раздела.

Подпрограмма pvm_exit сообщает локальному демону pvmd о том, что данный процесс завершает работу в PVM:

int ierr = pvm_exit(void) PVMFEXIT(IERR)

Процесс может продолжать работу и после этого вызова, но уже как последовательная программа. Обычно вызов подпрограммы pvm_exit размещается в конце программы. Параметр ierr содержит код завершения подпрограммы. Отрицательное значение соответствует ошибке. При выполнении подпрограмм pvm_mytid и pvm_exit могут возвращаться сообщения об ошибке

PvmSysErr.

Подпрограмма pvm_spawn запускает ntask копий исполняемого файла, имя которого содержится в первом параметре:

int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int

ntask, int *tids)

PVMFSPAWN(TASK, FLAG, WHERE, NTASK, TIDS, NUMT)

Параметр argv является указателем на массив аргументов задачи, который должен завершаться значением NULL. Если аргументов у задачи нет, NULL должно быть вторым элементом массива. Параметр flag используется для задания режимов выполнения запускаемого процесса, его значение равно сумме значений, приведенных в табл. 6.5.

Таблица 6.5. *Значения параметра flag подпрограммы pvm_spawn*

Значение	**Константа**	**Описание**
8	PvmTaskTrace	Генерировать данные трассировки. Обычно используется подпрограммой-монитором XPVM

16	PvmMppFront	Запускать задачи на фронтальном компьютере системы с массовым параллелизмом (MPP)
32	PvmHostCompl	Дополнение набора хостов в параметре where

Заметим, что приведенные в первом столбце таблицы числа являются просто степенями двойки, и суммирование значений равносильно установке двоичных разрядов. Имена, сопоставляемые численным значениям, содержатся в заголовочном файле pvm3/include/pvm3.h для программ на языке C и в файле pvm3/include/fpvm3.h для программ на языке FORTRAN.

Параметр where указывает, где следует запустить процесс. В качестве имени хоста можно использовать точку, которая обозначает локальный хост. После завершения вызова возвращается значение numt — количество фактически запущенных процессов. Если не удалось запустить ни одного процесса, этому параметру присваивается значение кода ошибки. В случае успешного запуска задач возвращается массив идентификаторов успешно запущенных задач tids, а в последние ntask — numt его элементов записываются коды ошибок. Ошибки возможны следующие: PvmBadParam, PvmNoHost, PvmNoFile, PvmNoMem, PvmSysErr И PvmOutOfRes. Подпрограмма pvm_spawn МОЖЕТ запускать задачи и на многопроцессорном компьютере. При работе на симметричном многопроцессорном компьютере следует выяснить особенности работы PVM на данной архитектуре.

В ОС UNIX подпрограмма pvm_spawn при запуске подчиненных задач передает им переменную окружения PVM_EXPORT, которая, в свою очередь, также содержит имена переменных, передаваемых подчиненным задачам.

Обмен сообщениями

В PVM передача сообщения выполняется за три шага. На первом создается буфер передачи (для этого используются подпрограммы pvm_initsend или pvm_mkbuf). Затем данные для сообщения упаковываются в буфер. Данные могут быть разнотипными, в этом случае используется многократный вызов подпрограмм pvm_pk* (в C) или pvmfpack (в FORTRAN) — по одному вызову для каждой порции данных. Последний шаг — отправка сообщения с помощью вызова подпрограммы pvm_send или другой (например, pvm_mcast).

Прием сообщения осуществляется вызовом подпрограммы блокирующего или неблокирующего приема. Данные из буфера приема распаковываются и могут быть

использованы в программе. Подпрограмма приема может принять:

- любое сообщение;
- сообщение с определенным тегом от любого процесса;
- определенное сообщение от определенного процесса.

Имеются функции проверки, с помощью которых можно определить, поступило ли уже сообщение.

Создание буфера

Буфер передачи создается с помощью подпрограммы pvm_initsend:

int bufid = pvm_initsend(int encoding)

PVMFINITSEND(ENCODING, BUFID)

Если программист использует только один буфер передачи, а это происходит чаще всего, ему достаточно подпрограммы pvm_initsend. Она вызывается перед упаковкой данных в буфер. Вызов подпрограммы pvm_initsend очищает буфер передачи, подготавливая его для упаковки нового сообщения. Параметр encoding позволяет указать метод кодирования данных при обмене в гетерогенной по форматам данных сети. Идентификатор нового буфера возвращается в параметре bufid.

В PVM используются следующие методы кодирования:

- FvmDataDefauit (или 0) — метод, используемый по умолчанию (XDR);
- pvmDataRaw (или 1) — передача данных без кодирования, которую следует использовать при работе в однородном кластере. В этом случае можно сэкономить на операциях кодирования при упаковке и распаковке данных;
- pvmDataInPlace (или 2) — данные остаются на своих местах в памяти, что позволяет уменьшить затраты на упаковку. Буфер передачи содержит только размер передаваемых данных и указатели на их адреса. При вызове pvm_send данные копируются непосредственно из памяти, из тех ячеек, которые они занимают. При этом количество операций копирования уменьшается, но программа не должна модифицировать данные в промежуток времени между упаковкой данных и моментом их передачи. Такую возможность можно использовать для того, чтобы применить операцию упаковки один раз, но передавать после этого данные неоднократно.

При выполнении данной подпрограммы возможны ошибки pvmBadParam и pvmNoMem. Коды ошибок возвращаются в параметре bufid.

Упаковка данных

Подпрограммы упаковки "укладывают", в общем случае, массив данных определенного типа в активный буфер передачи. При формировании одного сообщения .эти подпрограммы можно вызывать многократно, что позволяет передавать в одном сообщении разнотипные данные. Можно передавать и структуры языка С, упаковывая их индивидуальные элементы. Сложность сообщений в PVM поистине безгранична! Распаковка должна выполняться точно в том же порядке, что и упаковка.

Аргументами каждой подпрограммы являются указатели на первый элемент данных (cp). Параметр nitem задает общее количество элементов массива, подлежащих упаковке (это количество не обязательно совпадает с числом элементов массива); stride — параметр "гребенка". Значение stride = 1 соответствует упаковке непрерывного вектора, если stride = 2, упаковываются (и передаются) элементы первый, третий и т. д. Подпрограмма pvm_pkstr упаковывает символьную строку, которая заканчивается NULL, в этом случае, разумеется, параметры nitem и stride не нужны. В С имеется набор подпрограмм упаковки для каждого типа данных:

int ierr = pvmjpkbyte(char *cp, int nitem, int stride)

int ierr = pvm_pkcplx(float *cp, int nitem, int stride)

int ierr= pvm_pkdcplx(double *cp, int nitem, int stride)

int ierr = pvm_pkdouble(double *cp, int nitem, int stride)

int ierr = pvm_pkfloat(float *cp, int nitem, int stride)

int ierr = pvm_pkint(int *cp, int nitem, int stride)

int ierr = pvm_pklong(long *cp, int nitem, int stride)

int ierr = pvm_pkshort(short *cp, int nitem, int stride)

int ierr = pvm_pkstr(char *cp)

PVM содержит также подпрограмму упаковки pvm_packf, которая использует для описания схемы упаковки данных формат в духе оператора printf языка С.

В языке FORTRAN упаковка данных разных типов выполняется с помощью одной подпрограммы PVMFPACK:

PVMFPACK(WHAT, XP, NITEM, STRIDE, IERR)

Параметр xp задает первый элемент упаковываемого массива. Если упаковывается строка символов, ее длина должна быть задана значением параметра nitem. Параметр

(целочисленный) what задает тип данных. Допускаются значения, приведенные в табл. 6.6.

Коды ошибок, которые могут возвращаться при выполнении подпрограмм упаковки, — PvmNoMem И PvmNoBuf.

Таблица 6.6. *Допустимые значения параметра what в подпрограмме PVMFPACK*

Тип данных	**Числовой идентификатор**	**Тип данных**	**Числовой идентификатор**
STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REALS	6
INTEGER4	3	COMPLEX16	7

Подпрограммы передачи и приема данных

Передача сообщения из активного буфера выполняется подпрограммой

pvm_send:

int ierr = pvm_send(int tid, int msgtag)

PVMFSEND(TID, MSGTAG, IERR)

Здесь tid — идентификатор адресата, msgtag — неотрицательное целое значение тега сообщения, которое назначается программистом, а ierr — код завершения. В случае успешной передачи возвращается нулевое значение кода завершения, а в случае ошибки — значения pvmBadParam, FvmSysErr или PvmNoBuf.

В PVM используются несколько методов приема сообщений. Сообщение может быть принято любой подпрограммой приема, независимо от того, как оно было отправлено. Подпрограмма pvm_recv выполняет блокирующий прием сообщения от процесса с

указанным идентификатором (tid):

int bufid = pvm_recv(int tid, int msgtag)

PVMFRECV(TID> MSGTAG, BUFID)

Если значение параметра msgtag равно -1, оно играет роль "джокера" для тега сообщения, а "джокером" для источника является значение —1 параметра tid. Перед приемом сообщения буфер приема очищается. Код ошибки PvmBadParam возвращается, если неправильно указан идентификатор задачи.

Распаковка данных

Распаковка данных из принятого сообщения осуществляется подпрограммами:

int ierr = pvm_upkbyte(char *cp, int nitem, int stride)

int ierr = pvmupkcplx(float *cp, int nitem, int stride)

int ierr = pvm_upkdcplx(double *cp, int nitem, int stride)

int ierr = pvm_upkdouble(double *cp, int nitem, int stride)

int ierr = pvm_upkfloat(float *cp, int nitem, int stride)

int ierr = pvm_upkint(int *cp, int nitem, int stride)

int ierr. = pvm_upklong(long *cp, int nitem, int stride)

int ierr = pvm__upkshort (short *cp, int nitem, int stride)

int ierr = pvm_upkstr(char *cp)

Назначение параметров этих подпрограмм совпадает с назначением параметров соответствующих подпрограмм упаковки. В подпрограмме pvm_unpackf используется описание формата данных, подобное оператору printf языка С. В языке FORTRAN распаковка осуществляется подпрограммой:

PVMFUNPACK(WHAT, XP, NITEM, STRIDE, IERR)

Аргумент xp является массивом, в который выполняется распаковка. Целочисленный параметр what задает тип распаковываемых данных. При выполнении операции распаковки возможны ошибки: PvmNoData, pvmBadMsg и PvmNoBuf.

Коды ошибок

Ниже приводится список некоторых ошибок, коды которых возвращаются подпрограммами PVM:

- PvmSysErr (—14) — не отвечает демон pvmd;
- PvmBadParam (—2) — некорректное значение параметра;
- pvmNoHost (-6) — указанный хост не входит в состав виртуальной машины;
- PvmNoFile (—7) — не удается найти исполняемый файл программы;
- PvmNoMem (—10) — недостаточно памяти для запуска процесса;
- pvmOutOfRes (—27) — для запуска процесса не хватает ресурсов;
- PvmNoBuf (—15) — отсутствует активный буфер передачи сообщения;
- PvmNoData (—5) — попытка чтения после достижения конца файла. Обычно такая ошибка возникает, если из буфера считывается больше данных, чем там находится;
- PvmBadMsg (—12) — полученное сообщение не может быть раскодировано. Обычно связано с неправильным указанием параметров в подпрограммах упаковки.

Сходство и различие PVM и MPI

В среде разработчиков параллельных программ нередки споры о том, какая система лучше, MPI или PVM. Особенно часто возникают подобные обсуждения среди начинающих программистов. При сравнении обеих систем часто допускают ошибку, сравнивая спецификацию MPI с реализацией PVM. Для спецификаций и стандартов характерна меньшая гибкость, чем для их реализаций. В спецификациях MPI описываются возможности, которые рекомендуется включать в возможные реализации. Детальное сравнение обеих систем выходит за рамки нашей книги, его можно найти в специальных публикациях, посвященных параллельному программированию, но небольшой сравнительный обзор здесь уместен.

Так что же выбрать: MPI или PVM? Для того чтобы выбор был правильным и точным, следует учесть, что авторы обеих систем ставили перед собой разные цели. В отличие от PVM, которая с самого начала развивалась в рамках исследовательского проекта, спецификация MPI разрабатывается комитетом, состоящим из специалистов в области высокопроизводительных вычислений, которые активно работают как в исследовательских организациях, так и в промышленности. Отправной точкой деятельности комитета MPI Forum было то, что каждый производитель мультипроцессорных систем обычно создавал собственную систему обмена сообщениями. В этом случае не могло быть и речи о совместимости программного обеспечения. Совместимость мог бы обеспечить только стандарт для системы обмена сообщениями. Во главу угла деятельности MPI Forum была положена борьба за производительность, которую поддерживали производители суперкомпьютеров. С учетом этого не удивительно, что MPI позволяет почти всегда (но не всегда) обеспечить более высокую производительность на массивно-параллельных системах

(MPP), чем PVM.

MPI Forum не ориентировался на конкретные приложения, занимаясь созданием стандарта системы параллельного программирования. При этом в жертву стандартизации отчасти была принесена функциональность системы (потеря универсальности почти неизбежна при выработке любого стандарта).

Форум MPI декларировал и зафиксировал в спецификации ряд целей, некоторые из них приведены ниже.

- MPI должна быть библиотекой, пригодной для разработки параллельных прикладных программ, а не распределенной средой выполнения или распределенной операционной системой.
- MPI не ориентируется на многопоточные реализации, хотя и допускает их. Безопасность работы в многопоточной среде подразумевает отсутствие таких сущностей, как активный буфер или активное сообщение и т. д.
- Масштабируемость и корректность коллективных операций могут быть эффективно реализованы, только если группы процессов являются статическими.
- MPI должна быть модульной системой, что позволяет ускорить разработку переносимых параллельных библиотек. Следствием этого является то, что ссылки должны относиться к модулю, а не ко всей программе. Процессы-источники и адресаты должны задаваться рангом в группе, а не абсолютным идентификатором. Контекст обмена должен быть скрыт от программиста.
- MPI должна быть расширяемой системой, что дает возможность развивать ее для будущих применений. Это приводит к объектно-ориентированному стилю программирования в этой системе, хотя и без использования объектно-ориентированных языков. Для этого требуются операции с объектами, что является одной из причин относительно большого числа функций в MPI.
- MPI должна в какой-то степени поддерживать вычисления в гетерогенной среде (для этого введен такой объект, как MPi_Datatype), хотя и не требуется, чтобы все реализации системы были гетерогенными.
- MPI должно обеспечивать определенное поведение программ. Разработчики PVM изначально старались сделать программный и пользовательский интерфейс системы простым и понятным. Переносимости отдавалось предпочтение перед производительностью, исследования участников проекта сосредоточились на проблемах улучшения масштабируемости, надежности и поддержки гетерогенных вычислительных систем. В PVM принята довольно простая схема обмена сообщениями.

В процессе развития проекта сохранялась обратная совместимость PVM-программ, так чтобы уже работающие приложения оставались работоспособными и в новых версиях системы (по крайней мере, в рамках второй версии PVM). С другой стороны, можно выделить и характерные особенности MPI:

- большой набор подпрограмм двухточечного обмена;
- большой набор подпрограмм коллективного обмена;
- использование контекста обмена, который позволяет создавать надежные библиотеки параллельных подпрограмм;
- возможность создания и использования топологий обмена;
- возможность создания производных типов данных, позволяющих обмениваться сообщениями, которые содержат данные, расположенные в памяти не непрерывно.

В спецификации MPI-1 не была стандартизована процедура запуска MPI-программ в кластерах рабочих станций, что создавало проблемы с переносимостью. Одной из целей разработки спецификации MPI-2 было решение этой проблемы с помощью включения новых функций, среди которых:

- подпрограммы запуска процессов;
- односторонние операции коммуникации (put и get);
- неблокирующие операции коллективного обмена.

В MPI-2 к 128 функциям MPI-1 было добавлено еще 120 функций, что в результате дало гораздо более богатый набор подпрограмм, чем в PVM. Вместе с тем, пока в MPI нет тех полезных возможностей, которые предоставляет PVM. Это средства создания устойчивых к аппаратным сбоям приложений, динамическое определение доступных ресурсов и некоторые другие. Система PVM строится на основе концепции виртуальной машины, которая составляет базу поддержки гетерогенности, универсальности и других особенностей системы. MPI фокусируется на обмене сообщениями.

Интерфейс MPI должен был вобрать максимальное количество конструкций, связанных с обменом сообщениями и учесть особенности различных многопроцессорных систем так, чтобы программы могли выполняться на всех этих системах. Переносимость программ в MPI такова, что программа, написанная для одной архитектуры, может быть скопирована на другую архитектуру, откомпилирована и запущена на выполнение без модификации исходного кода. Такую переносимость поддерживает и PVM, но переносимость в PVM понимается в более широком смысле. Программы PVM также могут компилироваться на разных архитектурах и запускаться на выполнение, но при этом они могут обмениваться между собой данными. MPI-программа может выполняться на любой архитектуре, но архитектуры хостов должны совпадать. PVM-программа может работать на разнородном по архитектуре кластере. Спецификация MPI не запрещает поддержку гетерогенного исполнения, но и не включает его в разряд обязательных свойств. Это и понятно — сложно заставить одного производителя пожертвовать производительностью своей собственной системы ради совместимости с системой другого производителя.

Перед разработчиком системы обмена сообщениями всегда стоит дилемма: универсальность или производительность? Достичь максимальной производительности можно, используя особенности конкретной архитектуры, но это лишает систему универсальности.

В PVM производительность отчасти приносится в жертву гибкости и универсальности системы, которая должна поддерживать гетерогенность в максимальной степени. При выполнении обменов локально (на одном компьютере) или между компьютерами с одинаковой архитектурой, PVM использует собственные функции коммуникации для данной системы, как, впрочем, и MPI. Обмены между системами с разной архитектурой в PVM реализуются на основе стандартных сетевых протоколов. Для выбора механизма передачи сообщения PVM должна выполнить дополнительную работу, что снижает производительность системы.

Имеется различие между PVM и MPI в организации межъязыкового взаимодействия. В PVM возможно взаимодействие между программами, написанными на, языках C и FORTRAN. В MPI это взаимодействие не поддерживается. Причина — сложность согласования программных интерфейсов обоих языков.

Если говорить об управлении распределением ресурсов, следует отметить динамический характер PVM. Конфигурация виртуальной машины может изменяться динамически, как вручную, так и из прикладной программы. Это дает возможность программисту полнее реализовать динамически сбалансированную загрузку хостов, устойчивую работу программы. Потребности прикладной программы могут меняться -в процессе ее выполнения. Для эффективного использования вычислительного комплекса необходимо, чтобы система обеспечивала гибкий контроль применения и распределения ресурсов. Программа может часть времени работать в режиме последовательного выполнения, а в какие-то моменты времени ей могут потребоваться дополнительные процессоры для исполнения параллельной части кода. В этом случае не стоит сразу загружать множество процессоров одной задачей, гораздо выгоднее изменять загрузку параллельной системы в ходе выполнения программы. Подобные ситуации часто встречаются в программировании. В MPI нет средств, обеспечивающих такую динамику, это статическая по своей природе система. Статичность является в этом случае платой за производительность. Виртуальная машина в PVM является абстракцией, скрывающей в общем случае разнородную систему и позволяющую управлять ресурсами вычислительной системы. Вместе с тем, при необходимости, программист может отказаться от использования абстракции самого высокого уровня, переходя к более тонкой детализации (например, выбирая для запуска хост с определенной архитектурой). Спецификация MPI не поддерживает абстрагирования такого рода, оставляя программиста наедине с проблемой управления ресурсами.

MPI предлагает другой вид абстракции — топологию обменов, которая является надстройкой над множеством доступных вычислительных ресурсов. Топология

является логической, т. е. она обусловлена логической структурой алгоритма и схемой обмена данными в конкретной задаче. В соответствии с топологией выполняется переупорядочивание подзадач, относящихся к параллельной прикладной программе. В PVM такой возможности нет. Устойчивость работы является важным требованием, предъявляемым к программам с большим временем выполнения. Представьте себе: программа работает несколько дней и за пять минут до получения важного результата происходит сбой в системе. Ресурсы потрачены зря, а работа над исследовательским проектом отбрасывается назад. Чем больше время выполнения программы и чем больше для ее выполнения используется процессоров, тем важнее устойчивость системы, поскольку возрастает вероятность возникновения сбоя. Для обеспечения устойчивой работы система должна обладать средствами сбора информации о конфигурации, средствами определения аппаратных и программных сбоев и т. д.

В PVM имеется механизм уведомления об особых ситуациях, когда система сообщает задаче об изменении состояния виртуальной машины или об аварийном завершении процесса. Уведомлением является специальное сообщение и программист может предусмотреть собственную реакцию программы на то или иное событие. Можно привести и другой пример. Один из процессов может играть особую роль, являясь, например, сервером. Если хост, на котором выполняется эта задача, по какой-то причине перестает работать, программа, получив соответствующее уведомление, может запустить сервер на другом хосте. При изменении конфигурации виртуальной машины может быть предусмотрено и программное перераспределение ресурсов. Это обеспечивает эффективное выполнение параллельной PVM-программы.

В PVM имеется механизм уведомлений о прошедших событиях. Задача может с помощью этого механизма определить, например, завершил ли какой-то процесс работу. Может быть получено уведомление и о добавлении в виртуальную машину нового хоста. В MPI-1 таких возможностей нет, а в MPI-2 включена поддержка механизма уведомлений. Причина отсутствия в MPI средств обеспечения устойчивой работы и уведомлений об особых событиях— статический характер групп процессов MPI-1. В этом случае параллельная программа сразу же запускается на выполнение как коллектив взаимодействующих процессов, и если один из этих процессов аварийно завершит свою работу, завершится работа и всей программы. Статическая реализация групп следует естественным образом из системы приоритетов при разработке MPI — на первом месте производительность. В этом случае экономия достигается за счет отказа от средств мониторинга (наблюдения) за состоянием системы. Конфигурация группы процессов известна с самого начала.

Система PVM строится на основе концепции виртуальной машины, которая составляет базис поддержки гетерогенности, универсальности и других особенностей системы. MPI фокусируется на обмене сообщениями. При запуске программы MPI создается универсальный коммуникатор MPI_COMM_WORLD, в состав которого входят все запущенные процессы. При формировании новой группы процессов (области взаимодействия) выполняется синхронный вызов необходимых подпрограмм,

результатом которого является создание нового контекста обмена. В такой схеме создания группы не требуется участие специального сервера (или демона). Контекст разрушается при завершении работы хотя бы одного процесса из вновь созданного коммуникатора. При разработке спецификации MPI было решено, что генерация уникальных тегов контекстов обмена в разных группах может нанести ущерб производительности системы. Вместе с тем, наличие одинаковых тегов делает обмены в разных группах небезопасными из-за возможности их пересечения, поэтому и были введены интеркоммуникаторы, которые обеспечивают безопасный межгрупповой обмен, позволяя процессам "договориться" о контексте обмена. Напомним, что этот обмен всегда двухточечный.

Использование двух типов коммуникаторов в MPI является платой за отсутствие специальных демонов, обслуживающих обмен сообщениями. Коммуникаторы являются статическими объектами. Если по какой-то причине в работу программы привносится динамика (например, процесс выбывает из группы), ее работа может потерять детерминизм.

Группы в MPI и в PVM являются различными объектами, хотя и имеющими определенное сходство (например, в MPI адреса процессов определяются относительно группы, а в PVM они являются абсолютными и задаются в терминах идентификаторов задач).

В PVM, благодаря наличию демонов, несложно создать уникальный контекст обмена, что приводит к более простой и универсальной модели контекста. Имеются средства создания контекста и работы с ним. Эти средства похожи на средства MPI. При создании группы процессов она наделяется уникальным контекстом, и нет необходимости создавать такие дополнительные объекты, как коммуникаторы. Новые процессы в PVM могут использовать уже существующие контексты обмена, что позволяет им передавать сообщения и принимать их от членов группы. Безусловно, это полезное свойство. В MPI (в том числе и версии 2) изменение состава группы, прежде всего, удаление из нее процесса, разрушает коммуникатор и делает работу программы непредсказуемой.

Для сохранения обратной совместимости программ с предыдущими версиями PVM тег контекста не был включен в число параметров подпрограмм PVM. Этот объект "живет" в среде системы, а программа имеет к нему доступ и может его устанавливать.

В PVM версии 3.4 была введена концепция базового контекста. Текущий контекст наследуется дочерними задачами от родительского процесса. Если родительского процесса нет, т. е. задача является главной, она работает с базовым контекстом.

В PVM можно использовать *сервер имен,* роль которого играет демон системы. Сервером имен называют программу, которая возвращает информацию о заданном имени. Если независимо запускаются несколько программ, можно дать возможность

одной программе получить информацию о другой программе. Особенно это полезно в такой динамической системе, как PVM. В MPI-1 сервер имен просто не нужен, обходится без него и MPI-2.

Задача PVM, в том числе и демон PVM, может сформировать сообщение и поместить его на сервере имен, связав с сообщением определенный ключ. Ключом является строка, задаваемая программистом. Задачам, выполняющим поиск имени, передается сохраненное сообщение. Сообщение может содержать идентификаторы задач группы, контекст, связанный с ними, данные и другую информацию.

Задача может сохранить и сведения о собственнике именованного сообщения. Это сообщение удаляется при завершении работы процесса в PVM, при переинициализации виртуальной машины и т. д. Включение двух одинаково поименованных сообщений не допускается. Работа с сервером имен реализуется подпрограммами pvm_putinfo, pvm_recvinfo, pvm_getmboxinfo и pvm_delinfo.

Как в PVM 3.4, так и в MPI-2 есть возможность задать собственные обработчики сообщений. Программа может зарегистрировать функцию-обработчик, которая будет выполняться, только если задача получит соответствующее сообщение. Подпрограммы для работы с обработчиками сообщений: pvm_addmh и pvm_delmh.

В PVM нет средств параллельного ввода/вывода. Многие параллельные операции ввода/вывода являются коллективными и лучше всего определяются в терминах статических групп, механизм которых используется в MPI. В PVM затруднено программирование многопоточных приложений.

Так что же выбрать: MPI или PVM? Попытаемся дать несколько рекомендаций:

- если прикладная программа предназначена для выполнения на одном многопроцессорном компьютере, предпочтение следует отдать MPI, которая в этом случае позволит добиться наибольшей производительности;
- MPI-программа переносима на многопроцессорные машины других производителей, имеет гораздо более богатый набор подпрограмм обмена, чем PVM, поэтому MPI следует предпочесть, если предполагается использовать специальные режимы обмена, недоступные в PVM;
- MPI не поддерживает взаимодействие между программами, выполняющимися в различных реализациях MPI;
- PVM дает преимущество при работе в кластерах, особенно, если кластер разнородный. В PVM имеются средства управления распределением ресурсов и управления процессами;
- чем больше размер кластера и чем больше время выполнения программы, тем важнее устойчивость программ, которую в полной мере может обеспечить PVM. Управление процессами позволяет повысить эффективность выполнения

программ и в ситуации постоянного и динамического изменения загруженности хостов кластера. В MPI сложно (скорее, невозможно) создать устойчиво работающую в условиях аппаратных сбоев программу.

В настоящее время наблюдается определенное сближение обеих систем. Оно включает появление возможностей динамического управления процессами в MPI, статических групп и контекстов обмена в PVM.

Вопросы и задания для самостоятельной работы

1. Запустите PVM-консоль. Определите текущую конфигурацию виртуальной машины. Добавьте новый хост и вновь определите конфигурацию виртуальной машины. Удалите хост. Завершите работу с консолью командой quit. Запустите консоль заново и распечатайте конфигурацию. Как следует завершать работу консоли, если закончено выполнение параллельных программ PVM?

2. Запустите PVM-консоль. Определите текущую конфигурацию виртуальной машины. Добавьте новый хост. Завершите работу с консолью командой quit. Запустите консоль заново, но с другого хоста виртуальной машины, и распечатайте конфигурацию. Изменилась ли конфигурация?

3. Создайте хост-файл, включив в него компьютеры, входящие в состав вашего кластера. Запустите консоль и определите конфигурацию виртуальной машины. Прокомментируйте полученный результат.

4. Приведите примеры задач, для решения которых следует использовать MPI.

5. Приведите примеры задач, для решения которых следует использовать PVM.

6. Откомпилируйте программу, исходный текст которой содержится в листингах 6.5 и 6.6. Запустите ее на выполнение из консоли, а затем из командной строки.

7. Напишите программу, аналогичную приведенной в листингах 6.5 и 6.6, но на языке FORTRAN.

8. Напишите программу, аналогичную примеру из данной главы (см. листинги 6.5 и 6.6), но состоящую из одной программы. Откомпилируйте и запустите.

ГЛАВА 7.

Программирование с использованием PVM

В этой главе описываются средства PVM для управления процессами и виртуальной параллельной машиной. Мы познакомимся с различными методами передачи сообщений, буферизацией данных. Отдельный раздел посвящен работе с группами процессов. Несмотря на то, что данная глава завершает знакомство с PVM, мы сознательно оставляем "за кадром" некоторые возможности PVM, которые вряд ли понадобятся начинающему разработчику параллельных программ, но могут представлять интерес для создателя сложных приложений. Среди них средства работы с сервером имен и некоторые другие. Разобраться с соответствующими подпрограммами можно по справочным страницам и исходным текстам PVM. В заключение даются задания для самостоятельной работы.

Управление процессами

Управление любой системой складывается из "двух М" - мониторинга и модификации. *Мониторинг* — это наблюдение за текущим состоянием системы, в данном случае, сообщества процессов, относящихся к прикладной программе, сбор информации о системе. *Модификация* — здесь принудительное, инициированное программой, изменение состояния системы. Рассматривая управление процессами и виртуальной параллельной машиной в PVM, мы будем следовать этой схеме.

Подпрограммы для сбора информации о процессах

Подпрограмма pvm_parent возвращает идентификатор процесса, который запустил данный процесс, т. е. является по отношению к нему родительским:

int tid = pvm_parent(void) PVMFPARENT(TID)

Если процесс не имеет "родителя", возвращается специальное значение PvmNoParent, которое является, в действительности, кодом ошибки.

Информацию о задачах, которые выполняются в момент вызова на виртуальной машине PVM, можно получить с помощью подпрограммы pvm_tasks:

int ierr = pvm_tasks(int which, int *ntask, struct pvmtaskinfo **taskp)

PVMFTASKS(WHICH, NTASK, TID, PTID, DTID, FLAG, AOUT, IERR)

С помощью параметра which можно указать подпрограмме, о каких задачах следует сообщить информацию. Возможны следующие варианты:

- при нулевом значении параметра информация возвращается обо всех задачах, выполняющихся на виртуальной машине;
- если указан идентификатор демона dtid, информация возвращается обо всех задачах на соответствующем хосте (хосте, на котором работает данный демон);
- если указан идентификатор задачи, информация возвращается об этой конкретной задаче.

Параметр ntask содержит количество задач, а параметр taskp является указателем на массив структур pvmtaskinfo, который содержит ntask элементов. В каждой структуре pvmtaskinfo содержатся: идентификатор задачи tid, идентификатор задачи демона pvmd (dtid), идентификатор родительского процесса ptid, флаг состояния flag, имя исполняемого файла и системный идентификатор процесса pid (его значение зависит от операционной системы).

Структура pvmtaskinfo устроена следующим образом:

struct pvmtaskinfo{ int ti_tid; int ti_ptid; int ti_host; int ti_flag; char * ti_a_out; int ti_pid; }

Если задача была запущена вручную, имя исполняемого файла для нее не может быть определено системой. В этом случае значения соответствующим полям структуры pvmtaskinfo не присваиваются.

В программах на языке FORTRAN один вызов возвращает информацию лишь об одной задаче. Если параметр where = 0 и выполнено ntask вызовов подпрограммы pvm_tasks, информация будет получена обо всех задачах.

Возможные ошибки выполнения: PvmBadParam, PvmSysErr и PvmNoHost.

Для получения информации о процессе с идентификатором tid используется подпрограмма pvm_pstat:

int status = pvm_pstat(tid)

PVMFPSTAT(TID, STATUS)

Результат выполнения — значение параметра status:

- Pvmok — задача выполняется;
- PvmNoTask — задача не выполняется;
- pvmBadParam — неправильно задан идентификатор tid.

В листинге 7.1 приводится программа вычисления скалярного произведения двух векторов. Есть два одномерных массива (векторы) *x* и *y* по *N* элементов каждый. Необходимо вычислить значение:

$$p = \sum_{i=1}^{N} x_i y_i .$$

Эта задача идеально подходит для распараллеливания, поэтому она включена в число примеров, поставляемых вместе с PVM. Программа работает следующим образом. После того, как пользователь программы задаст количество процессов, которые будут заниматься вычислением скалярного произведения, каждый массив разбивается на равные части. Затем запускается необходимое количество процессов, и фрагменты массивов пересылаются подчиненным задачам. Вместе с подмассивами пересылается и количество элементов в подмассиве. Если разбить на равные части не удается, часть массивов оставляет себе главный процесс. Затем каждая задача вычисляет часть общей суммы и полученное в результате число передает главному процессу. Главный процесс суммирует полученные значения и формирует окончательный результат.

Листинг 7.1. Программа вычисления скалярного произведения векторов

```
PROGRAM PSDOT

INCLUDE "/HOME/PUB/PVM3/INCLUDE/FPVM3.H"

EXTERNAL PVMFMYTID, PVMFPARENT,

PVMFSPAWN, PVMFEXIT, PVMFINITSEND

EXTERNAL PVMFPACK, PVMFSEND,

PVMFRECV, PVMFUNPACK

REAL SDOT

EXTERNAL SDOT
```

```
INTRINSIC MOD

INTEGER MAXN

PARAMETER (MAXN = 800000)

INTEGER N, LN, MYTID, NPROCS, IBUF, IERR

INTEGER I, J, K

REAL LOOT, GDOT

INTEGER TIDS(0:9) REALX(MAXN), Y(MAXN)

CALL PVMFMYTID(MYTID) CALL PVMFPARENT(TIDS(0))

IF (TIDS(0).EQ.PVMNOPARENT) THEN

PRINT *, 'HOW MANY PROCESSES SHOULD

PARTICIPATE (1-10)?' READ(*, *) NPROCS

PRINT *, 'ENTER THE LENGTH OF VECTORS

TO MULTIPLY' READ(*, *) N TIDS(O) = MYTID

IF (N.GT.MAXN) THEN PRINT *, 'N TOO LARGE' STOP END IF

J = N / NPROCS LN = J + MOD{N, NPROCS)

I = LN + 1 DO MN = 1, N X(MN) = 1. Y(MN) = 1. ENDDO

С ЦИКЛ ЗАПУСКА ПОДЧИНЕННЫХ ЗАДАЧ

И РАСПРЕДЕЛЕНИЯ МЕЖДУ НИМИ РАБОТЫ

DO K = 1, NPROCS - 1

CALL PVMFSPAWN('DOT', 0, 'ANYWHERE', 1,

TIDS(K), IERR) IF (IERR.NE. 1) THEN

PRINT *, 'ERROR, COULD NOT SPAWN PROCESS I',

K CALL PVMFEXIT(IERR) STOP END IF

CALL PVMFINITSEND(PVMDEFAULT, IBUF)
```

```
CALL PVMFPACK(INTEGER4, J, 1, 1, IERR)
CALL PVMFPACK(REAL4, X(I), J, 1, IERR)
CALL PVMFPACK(REAL4, Y(I), J,'I, IERR)
CALL PVMFSEND(TIDS(K), 0, IERR) I = I + J END DO
C КОНЕЦ ЦИКЛА ЗАПУСКА ПОДЧИНЕННЫХ
ЗАДАЧ И РАСПРЕДЕЛЕНИЯ РАБОТЫ С
ГЛАВНАЯ ЗАДАЧА ВЫЧИСЛЯЕТ СВОЮ ЧАСТЬ
СКАЛЯРНОГО ПРОИЗВЕДЕНИЯ
GDOT = SDOT(LN, X, 1, Y, 1)
C ПРИЕМ РЕЗУЛЬТАТОВ ВЫЧИСЛЕНИЙ ОТ ПОДЧИНЕННЫХ ЗАДАЧ
DO К = 1, NPROCS - 1 CALL PVMFRECV(-1, 1, IBUF)
CALL PVMFUNPACK(REAL4, LOOT, 1, 1, IERR) GDOT = GDOT + LOOT
END DO
C КОНЕЦ ПРИЕМА РЕЗУЛЬТАТОВ ВЫЧИСЛЕНИЙ
ОТ ПОДЧИНЕННЫХ ЗАДАЧ С ВЫВОД КОНЕЧНОГО РЕЗУЛЬТАТА
PRINT *, '<X,Y> = ', GDOT ELSE C ЭТОТ ФРАГМЕНТ
КОДА ОТНОСИТСЯ К ПОДЧИНЕННОЙ ЗАДАЧЕ
CALL PVMFRECV (TIDS (0) , 0, IBUF)
CALL PVMFUNPACK(INTEGER4, LN, 1, 1,- IERR)
CALL PVMFUNPACK(REAL4, X, LN, 1, IERR)
CALL PVMFUNPACK(REAL4, Y, LN, 1, IERR)
C ВЫЧИСЛЕНИЕ ЧАСТНОГО ПРОИЗВЕДЕНИЯ И
ПЕРЕДАЧА РЕЗУЛЬТАТА С ГЛАВНОЙ ЗАДАЧЕ
LOOT = SDOT(LN, X, 1, Y, 1)
```

CALL PVMFINITSEND(PVMDEFAULT, IBUF)

CALL PVMFPACK(REAL4, LOOT, 1, 1, IERR)

CALL PVMFSEND(TIDS(0), 1, IERR) END IF

C КОНЕЦ ФРАГМЕНТА КОДА ОТНОСЯЩЕГОСЯ К

ПОДЧИНЕННОЙ ЗАДАЧЕ CALL PVMFEXIT(IERR) STOP END

C ФУНКЦИЯ ВЫЧИСЛЕНИЯ СКАЛЯРНОГО ПРОИЗВЕДЕНИЯ

FUNCTION SDOT(LN, X, N1, Y, MI)

REAL X(LN),Y(LN)

S = 0.

DO I = 1, LN

S = S + X(I) * Y(I) END DO SDOT = S RETURN END

Вычисление частного скалярного произведения выполняет функция SDOT. Обратите внимание на то, что исполняемый файл программы должен быть сохранен под именем dot. Разберите самостоятельно работу этой программы и попробуйте ее выполнить, проверив результат вычислений.

Изменение состояния процессов

Основными операциями изменения состояния процессов являются их запуск и завершение работы. С подпрограммой запуска процессов pvm_spawn мы познакомились в *главе 6.* С помощью подпрограммы pvm_kiii можно завершить работу задачи с указанным идентификатором tid:

int ierr = pvm_kill(int tid) PVMFKILL(TID, IERR)

Не следует завершать работу процесса, из которого выполняется данный вызов. Эта подпрограмма не предназначена для "самоубийства" (для этого есть подпрограмма pvm_exit). При работе в кластере она посылает сигнал SIGTERM заданному процессу, а в мультипроцессорной системе используется свой, машинно-зависимый способ завершения. При выполнении этой подпрограммы возможны ошибки: PvmBadParam, если неправильно задан идентификатор задачи, и pvmSysErr, если не отвечает демон pvmd.

В ОС UNIX основным средством управления процессами являются *сигналы.* Сигналы посылаются процессам как операционной системой, так и прикладной программой.

Чаще всего реакцией процесса на сигнал является завершение его работы, хотя возможны и другие варианты. Подпрограмма pvm_sendsig посылает сигнал PVM-процессу с указанным идентификатором tid:

int ierr = pvm_sendsig(int tid, int signum)

PVMFSENDSIG(TID, SIGNUM, IERR)

Сигнал задается его номером signum. Данную подпрограмму следует использовать осторожно, как, впрочем, и любую другую подпрограмму, изменяющую состояние процесса. Перечень сигналов можно найти в руководствах по операционной системе UNIX.

Приведем пример передачи самого "сильного" сигнала безусловного завершения процесса:

ierr = pvm_sendsig(tid, SIGKILL);

Ошибки при выполнении этой подпрограммы:

PvmSysErr и PvmBadParam.

Полезной может оказаться подпрограмма pvm__notify, которая активизирует режим оповещения вызывающего процесса о наступлении определенных событий:

int ierr = pvm_notify(int what, int msgtag, int cnt, int tids)

PVMFNOTIFY(WHAT, MSGTAG, CNT, TIDS, IERR)

У этой подпрограммы следующие ключи what:

- *P*vmTaskExit — оповещать о завершении работы задачи;
- *P*vmHostDelete — оповещать, если хост удаляется из виртуальной машины (или происходит сбой);
- PvmHostAdd — оповещать о добавлении в виртуальную машину нового хоста.

При наступлении определенных событий система посылает вызывающему процессу сообщения. Эти сообщения маркируются тегами msgtag, которые задаются программистом. Массив tids определяет процесс (хост), к которому относится действие ключей TaskExit или HostDeiete. При использовании ключа HostAdd массив пустой. Сообщение об аварии хоста придет в любом случае. Полезность этой программы очевидна при разработке прикладных программ, устойчивых к аппаратным сбоям, когда программист готов взять на себя ответственность за обработку особых ситуаций в системе.

По умолчанию стандартные потоки вывода и протокол ошибок запущенных процессов записываются в файл /tmp/pvml.UID. С помощью подпрограммы pvm_catchout вывод запущенных задач может быть "перехвачен вызывающим процессом:

int ierr = pvm_catchout(FILE *ff)

PVMFCATCHOUT(ONOFF, IERR)

В результате вызова вывод будет направляться данному процессу и записываться в указанный файл (в C файл задается его дескриптором) или в стандартный вывод (в языке FORTRAN). Параметр ONOFF включает или отключает режим перехвата вывода. Перехватывается и вывод процессов, запущенных дочерними задачами. Каждая порция данных маркируется информацией об их источнике, причем текстовый маркер добавляется в начало вывода и в его конец:

[t80008] BEGIN

[t80008] Earth mass is 3.14

[t80008] END

Маркировка применяется для того, чтобы пользователь мог разобраться в том, откуда пришло соответствующее сообщение. Операция захвата является блокирующей в том смысле, что даже после вызова подпрограммы pvm_exit процессом, собирающим вывод от дочерних задач, он не завершит своей работы в PVM, пока не получит сообщения от всех задач. Если блокировка не нужна, перед вызовом pvm_exit следует поставить вызов pvm_catchout(0), который отключает режим перехвата. Подпрограмма pvm_catchout всегда завершается успешно (с кодом Pvmok).

Работа с буфером

Подпрограммы, которые мы описываем в данном разделе, требуются только в том случае, когда программист собирается использовать в одной программе несколько буферов. Сразу заметим, что это требуется не так часто.

В PVM каждый процесс имеет один *активный буфер* передачи и приема. Все остальные буферы, если они есть, считаются *пассивными.* Программист, при необходимости, может сделать любой из них активным. Операции упаковки, передачи, приема и распаковки применяются только к активному буферу.

Подпрограмма pvm_mkbuf создает новый пустой буфер (обычно это буфер передачи) и определяет для него метод кодирования данных (encoding):

int bufid = pvm_mkbuf(int encoding)

PVMFMKBUF(ENCODING, BUFID)

Возвращается идентификатор буфера bufid. Отрицательное значение идентификатора буфера говорит о том, что при выполнении подпрограммы произошла ошибка. Использованный буфер должен быть удален, а память -освобождена, поэтому подпрограмма pvm_mkbuf используется в паре с подпрограммой pvm freebuf, которая освобождает буфер с идентификатором

bufid:

int ierr = pvm_freebuf(int bufid)

PVMFFREEBUF(BUFID, IERR)

Удалять буфер следует после передачи сообщения. Создается буфер для нового сообщения вызовом подпрограммы pvm_mkbuf. Подпрограммы создания и удаления буфера не нужны, если при передаче сообщения используется подпрограмма pvm_initsend, выполняющая все необходимые действия.

Буфер приема автоматически создается в момент приема сообщения подпрограммами pvm_recv и pvm_nrecv. Его не надо специально удалять, если только он не был сохранен обращением к подпрограмме pvm_setrbuf. Еще раз напомним, что работа с несколькими буферами -- явление довольно редкое, а при работе с одним буфером достаточно подпрограммы

pvm_initsend.

Вероятные ошибки выполнения pvm_freebuf: PvmBadParam и PvmNoSuchBuf

(неправильное значение bufid). При создании буфера возможны ошибки: PvmBadParam (неправильно указан метод кодирования) и PvmNoMem (для создания буфера не хватает памяти).

Подпрограмма pvm_getsbuf возвращает идентификатор активного буфера передачи, а pvm_getrbuf — идентификатор активного буфера приема (bufid):

int bufid = pvm_getsbuf(void) PVMFGETSBUF(BUFID)

int bufid = pvm_getrbuf(void) PVMFGETRBUF(BUFID)

Нулевое значение bufid возвращается, если буфер отсутствует. При выполнении обеих подпрограмм коды ошибок не формируются.

Если в программе используются хотя бы два буфера, один из них должен играть выделенную роль. Операции упаковки, передачи и приема должны применяться к

этому, активному буферу. Необходим также механизм переключения между буферами. Подпрограммы:

int oldbuf = pvm_setsbuf (int bufid)

PVMFSETRBUF(BUFID, OLDBUF)

int oldbuf = pvm_setrbuf(int bufid)

PVMFSETRBUF(BUFID, OLDBUF)

переключают активный буфер передачи и приема соответственно. Каждая из них активизирует буфер с идентификатором bufid, сохраняя при этом состояние предыдущего активного буфера, а также его идентификатор (в переменной oldbuf). При успешно выполненное приеме автоматически создается новый активный буфер приема. Если содержимое предыдущего буфера приема не было распаковано, оно может быть сохранено для дальнейшей работы.

Если в подпрограммах pvm_setsbuf или pwi_setrbuf параметру bufid присвоено значение 0, текущий буфер сохраняется, но активный буфер не определен. Эту особенность PVM можно использовать для сохранения текущего состояния сообщений с тем, чтобы, например, математическая библиотека, использующая сообщения PVM, не влияла на состояние буферов программы. После завершения работы с такой библиотекой буферы вновь можно сделать активными.

Сообщения можно переадресовывать без переупаковки с помощью подпрограмм создания буфера, например:

bufid = pvm_recv(src, *tag);* oldid = pvm_setsbuf(bufid);

ierr = pvm_send(dst, tag); ierr = pvm_freebuf(oldid);

При выполнении подпрограммы pvm_setrbuf возможны ошибки: PvmBadParam

(неправильный идентификатор буфера), pvmNoSuchBuf (переключение в несуществующий буфер).

Подпрограмма pvm_bufinfo возвращает информацию о буфере сообщения:

int ierr = pvm_bufinfo(int bufid, int *bytes, int *msgtag, int *tid)

PVMFBUFINFO(BUFID, BYTES, MSGTAG, TID, IERR)

Ее параметры:

- bufid — идентификатор буфера;

- bytes — длина сообщения в байтах, выходной параметр;
- msgtag — тег сообщения, выходной параметр;
- tid — идентификатор источника сообщения, выходной параметр;
- ierr — код завершения.

Обычно подпрограмма pvm_bufinfo используется для получения информации о размере и источнике последнего принятого сообщения в случае, когда задача может принимать любые сообщения. При выполнении данного вызова возможны ошибки PvmNoSuchBuf и PvmBadParam.

Передача и прием сообщений

Основными подпрограммами передачи и приема сообщений являются pvm_send и pvm_recv, с которыми мы познакомились в предыдущем разделе. Здесь мы рассмотрим дополнительные возможности обмена сообщениями в PVM. Подпрограмма pvm_psend упаковывает и передает массив указанного типа задаче с заданным идентификатором и делает это за одно обращение к ней:

int ierr = pvm_psend(int tid, int msgtag, void *buf, int len, int datatype)

PVMFPSEND(TID, MSGTAG, BUF, LEN, DATATYPE, IERR)

Параметры подпрограммы:

- tid — идентификатор адресата сообщения;
- msgtag — тег сообщения, задается программистом;
- buf — указатель на буфер передачи;
- len — размер буфера (значение должно быть кратно размеру используемого типа данных);
- datatype — тип данных;
- ierr — код завершения.

Прием сообщения, отправленного с помощью подпрограммы pvm_psend, могут выполнить подпрограммы pvm_precv, pvm_recv, pvm_trecv и pvm_nrecv.

Передача с помощью подпрограммы pvm_psend является неблокирующей. Типы в версии подпрограммы для языка FORTRAN такие же, как и в подпрограмме PVMFPACK. В C тип данных задается идентификатором, список которых приведен в табл. 7.1.

Таблица 7.1. *Соответствие типов данных в PVM и в C*

Идентификатор типа	**Тип в C**

в PVM	
PVM_STR	String
PVM BYTE	Byte
PVM_SHORT	Short
PVM INT	Int
PVM FLOAT	Real
PVM__CPLX	Complex
PVM DOUBLE	Double
PVM_DCPLX	Double complex
PVM_LONG	Long integer
PVM USHORT	Unsigned short int
PVM_UINT	Unsigned int
PVM_ULONG	Unsigned long int

Ошибки, которые могут возникнуть при выполнении подпрограммы

pvm_psend: PvmBadParam (неправильные значения tid или msgtag) и PvmSysErr (не отвечает демон pvmd).

Подпрограмма pvm_nrecv выполняет неблокирующий прием сообщения от процесса с идентификатором tid:

int bufid = pvm_nrecv(int tid, int msgtag)

PVMFNRECV(TID, MSGTAG, BUFID)

Значение tid = -i играет роль "джокера". Если сообщение еще не пришло, возвращается значение bufid = o, а если пришло, оно помещается в новом активном буфере приема, который замещает ранее существовавший буфер (если он, конечно, существовал). Параметр msgtag содержит тег сообщения, а значение —1 соответствует любому тегу (т. е. является "джокером"). Подпрограмму pvm_nrecv можно вызывать многократно для проверки прибытия сообщения. Если сообщение прибыло, подпрограмма pvm_nrecv поместит его в новый активный буфер приема и вернет идентификатор этого буфера. Отрицательное значение bufid говорит об ошибке.

Примеры для C:

tid = pvm_parent();

msgtag = 1;

arrived = pvm_nrecv(tid, msgtag);

if (arrived > 0)

ierr = pvm_upkint(buf, nitems, stride);

и для FORTRAN:

CALL PVMFNRECV(-1, MSGTAG, ARRIVED)

IF (ARRIVED.GT.0) THEN

CALL PVMFUNPACK(INTEGER4, TIDS, NITEMS, STRIDE, IERR)

ENDIF

Возможны следующие ошибки выполнения данной подпрограммы: PvmBadParam (неправильно указаны идентификатор источника или тег сообщения) И PvmSysErr.

При организации неблокирующего обмена необходима подпрограмма, которая проверяет, прибыло ли уже сообщение, но не принимает его. В PVM такой подпрограммой-пробником является pvm_probe:

int bufid = pvm_probe(int tid, int msgtag)

PVMFPROBE(TID, MSGTAG, BUFID)

Эта подпрограмма возвращает нулевое значение bufid, если сообщение от задачи с идентификатором tid еще не прибыло. Прием сообщения в любом случае не выполняется, возвращается лишь идентификатор буфера для прибывшего сообщения. Данная подпрограмма также может вызываться неоднократно. Кроме того, впоследствии может быть вызвана подпрограмма pvm_bufinfo для получения информации о сообщении еще до его приема. Значения —1 для идентификатора источника и тега сообщения являются "джокерами".

Примеры для C:

```
arrived = pvm_probe(tid, msgtag);

if (arrived > 0)

ierr = pvm_bufinfo(arrived, &len, &tag, &tid);
```

и для FORTRAN:

```
CALL PVMFPROBE(-1, MSTAG, ARRIVED)

IF (ARRIVED.GT. 0) THEN

CALL PVMFBUFINFO(ARRIVED, LEN, TAG, TID, IERR)

ELSE

ENDIF
```

Возможные ошибки выполнения данной подпрограммы: PvmBadParam и PvmSysErr.

Случается так, что сообщение по какой-то причине никогда не достигнет адресата. В этом случае использование блокирующего приема нежелательно, т. к. он может заблокировать процесс (а с ним и всю программу) навечно. Для того чтобы избежать таких ситуаций, можно использовать подпрограм-

му pvm_trecv, которая позволяет программисту указать время ожидания сообщения, по истечении которого выполнение процесса будет продолжено независимо от того, получено сообщение или нет. В течение заданного времени выполнение вызывающего процесса блокируется. Предельный случай большого времени ожидания соответствует блокирующему приему, а нулевое время ожидания — неблокирующему приему. Подпрограмма pvm_trecv выполняет прием сообщений с ограниченным временем ожидания и занимает промежуточное положение между подпрограммами блокирующего и неблокирующего приема:

```
int bufid = pvm_trecv(int tid, int msgtag, struct timeval *tmout)
```

PVMFTRECV(TID, MSGTAG, SEC, USEC, BUFID)

Ее параметры:

- tid — идентификатор источника сообщения;
- msgtag — тег сообщения;
- tmout — время ожидания сообщения. Если сообщение не будет получено в течение этого времени, работа подпрограммы завершается;
- sec, usec — целочисленные значения, определяющие время ожидания, секунды и миллисекунды;
- bufid — идентификатор буфера сообщения (отрицательное значение возвращается при ошибке).

Значение —1 параметров msgtag и tid играет роль "джокера".

В C время ожидания содержится в структуре tmout с полями tv_sec и tv_usec. В версии для языка FORTRAN используются два параметра: sec и usec. Если оба равны нулю, pvm_trecv ведет себя аналогично подпрограмме pvm_nrecv, выполняющей проверку прибытия сообщений и завершающей свою работу сразу же, даже если сообщений нет. В C передача нулевого указателя в tmout приводит к поведению, аналогичному pvm_recv, т. е. блокирующему. Такой же эффект дает в языке FORTRAN значение sec = -1.

Примеры использования этой подпрограммы в языке C:

struct timeval tmout;

if ((bufid = pvm_trecv(tid, msgtag, &tmout)) >0)

{ pvm_upkint(arrayl, nitemsl, stridel);

pvm_upkint(param, nitems2, stride2);

pvm_upkfloat(array2, nitemsS, strides); }

и в языке FORTRAN:

CALL PVMFRECV(-1, 4, 60, 0, BUFID)

IF (BUFID.GT.0) THEN

CALL PVMFUNPACK(INTEGER4, TIDS, NITEMS1,

STRIDE1, IERR) CALL PVMFUNPACK(REAL8, ARR,

NITEMS2, STRIDE2, IERR) ENDIF

При выполнении данной подпрограммы возможны ошибки:

PvmBadParam (при неправильном значении tid или msgtag < -1) и PvraSysErr.

Подпрограмма pvm_precv объединяет функции блокирующего приема и распаковки:

int ierr = pvm_precv(int tid, int msgtag, void *buf, int len,

int datatype, int *rtid, int *rtag, int *rlen)

PVMFPRECV(TID, MSGTAG, BUF, LEN,

DATATYPE, RTID, RTAG, RLEN, IERR)

Она принимает сообщение прямо в буфер.

Параметры:

- tid — идентификатор задачи-источника сообщения;
- msgtag — тег сообщения;
- buf — указатель на буфер, в который будет выполняться прием сообщения;
- len — размер буфера (должен задаваться числом, кратным размеру используемого типа данных);
- datatype — тип данных;
- rtid — фактический идентификатор источника;
- rtag — фактический тег сообщения;
- rlen — фактический размер сообщения;
- ierr — статус завершения.

Подпрограмма pvm_precv может выполнять пр ием сообщений, отправленных подпрограммами pvm_psend, pvm_send, pvm_mcast и pvm_bcast.

Возможные ошибки выполнения: PvmBadParam и PvmSysErr.

Подпрограмма pvm_perror выводит на экран статус последнего обращения к подпрограмме PVM:

int ierr = pvm_perror(char *msg)

PVMFPERROR(MSG, IERR)

У этой подпрограммы два параметра:

- msg - символьная строка, которая присоединяется к сообщению об ошибке и задается программистом. В этом сообщении может содержаться, например, идентификатор задачи, из которой последовал вызов или другая информация;
- ierr — статус завершения вызова.

Все сообщения из стандартных потоков вывода и ошибок записываются в файл /tmp/pvml.UID на хосте с главным демоном pvmd. Данная подпрограмма не возвращает коды ошибок (ошибок здесь просто не может быть).

Управление виртуальной машиной

Управление виртуальной машиной, как обычно, складывается из двух частей: сбора информации о состоянии машины в целом и отдельных хостов, входящих в ее состав, а также изменения конфигурации или режима работы виртуальной машины.

Сбор информации о состоянии виртуальной параллельной машины

Информацию о виртуальной машине можно получить с помощью подпрограммы pvm_config:

int ierr = pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)

PVMFCONFIG(NHOST, NARCH, DTID, NAME, ARCH, SPEED, IERR)

Возвращаются: количество хостов в виртуальной машине nhost, количество различных форматов хранения данных narch. Параметр hostp является указателем на заданный пользователем массив структур pvmhostinfo. Размер этого массива не менее nhost. Каждая структура pvmhostinfo содержит идентификатор задачи для демона pvmd (dtid), имя хоста, наименование архитектуры и относительное быстродействие центрального процессора (по умолчанию 1000). Структура pvmhostinfo устроена следующим образом:

struct pvmhostinfo { int hi_tid; char *hi_name; char *hi_arch; int hi_speed; }

В языке FORTRAN один вызов этой подпрограммы возвращает информацию об одном хосте. Для получения информации обо всей виртуальной машине требуется многократный вызов подпрограммы — по одному вызову на каждый хост. При этом формируется значение одного элемента массива

HOST:

DO I=1, NHOST

CALL PVMFCONFIG(NHOST, NARCH, DTID(I),

HOST(I), ARCH(I), SPEED(I), IERR) ENDDO

Неудобство такого подхода заключается в том, что в процессе опроса состояние уже опрошенных хостов может измениться, но программа об этом не узнает до нового цикла опроса. При выполнении этой подпрограммы возможна только ошибка PvmSysErr, если не отвечает локальный демон.

Информацию о состоянии определенного хоста, входящего в состав виртуальной машины, можно получить с помощью подпрограммы pvm_mstat:

int mstat = pvm_mstat(char *host) PVMFMSTAT(HOST, MSTAT)

Хост задается его именем host — это строка символов, а результатом является целое значение mstat:

- PvmOk — хост работает нормально;
- PvmNoHost — хост не входит в состав виртуальной машины;
- PvmHostFail — хост недостижим, скорее всего, по причине аппаратного сбоя.

Все эти значения являются, в действительности, кодами ошибок PVM. С помощью подпрограммы pvm_mstat можно организовать мониторинг состояния виртуальной машины с тем, чтобы при возникновении аварийных ситуаций можно было из прикладной программы изменить конфигурацию виртуальной машины.

Подпрограмма pvm_tidtohost возвращает dtid — идентификатор задачи демона pvmd, работающего на том же хосте, что и процесс с указанным идентификатором tid:

int dtid = pvm_tidtohost(int tid)

PVMHTIDTOHOST(TID, DTID)

Эта подпрограмма используется для определения, на каком компьютере выполняется процесс с заданным идентификатором. Если неправильно задан параметр tid, возвращается код ошибки PvmBadParam.

Пример использования подпрограмм сбора информации приведен в листингах 7.2 и 7.3. Программа-пример построена по схеме "главная/подчиненная" задача. После запуска она определяет количество хостов, входящих в состав виртуальной машины, а затем на каждом из них запускает по три процесса. Исполняемый файл подчиненной задачи должен быть сохранен с именем slave. Этот пример взят из коллекции примеров, поставляемых вместе с PVM, и читателю полезно разобраться, как работает эта программа.

Листинг 7.2. Главная программа примера программы master-slave

```
#include <stdio.h>
#include "...pvm3/include/pvm3.h"
main ()
{
int mytid;
int tids[32] ;
int n, nproc, numt, i, who, msgtype, nhost, narch;
float data [100] , result [32];
struct pvmhostinfo *hostp;
mytid = pvm_mytid() ;
pvm_config(&nhost, &narch, Shostp) ;
nproc = nhost * 3;
if (nproc > 32) nproc = 32;
printf ("Spawning %d worker tasks ", nproc);
numt=pvm_spawn( "slave", (char**)0, 0, "", nproc, tids) ; if (numt < nproc) (
printf ("\n Trouble spawning slaves.
Aborting. Error codes are:\n" for(i=numt; Knproc; i++)
( printf ("TID %d %d\n", i, tidsti]); }
for(i =0; i < numt; i pvm_kill(tids[i] ) ; }
pvm_exit ( ) ; exit(l) ; } printf ("SUCCESSFULXn") ;
n = 100;
for (i = 0; i < n; i++) {
data[i] =1.0;
}
```

```
pvm_initsend ( PvmDataDe fault) ; pvm_pkint (&nproc, 1, 1);
pvm_pkint (tids, nproc, 1) ; pvm_pkint (&n, 1, 1);
pvm_pkf loat (data, n, 1); pvm_mcast (tids, nproc, 0);
msgtype = 5;
for(i = 0; i < nproc; i++) ( pvm_recv ( -1 , msgtype ) ;
pvm_upkint (&who, 1, 1) ; pvm_upkfloat (&result [who] , 1, 1)
printf("I got %f from %d; ", result[who], who);
if (who == 0)
printf(" (expecting %f)\n", (nproc - 1) * 100.0);
else
printf ("(expecting %f)\n", (2 * who - 1) * 100.0);
}
pvm_exit ( ) ;
}
```

Листинг 7.3. Подчиненная программа примера программы master-slave

```
# include <stdio.h>
#include ". . .pvm3/include/pvm3.h"
main ()
{
int mytid;
int tids[32] ;
int n, me, i, nproc, master, msgtype;
float data [100], result, sum;
float work ( ) ;
```

```
mytid = pvm_mytid ( ) ;

msgtype = 0; pvm_recv ( -1 , msgtype ) ;

pvm_upkint (&nproc, 1, 1);

pvm_upkint (tids, nproc, 1);

pvm_upkint ( &n , 1 , 1 ) ;

pvm_upkf loat (data/ n, 1);

}

for(i = 0; i < nproc; i

if (mytid == tids[i]){ me = i; break; }

sum = 0.0;

for (i =0; i < n; i++) { sum += me * data[i];

}

pvm_initsend(PvmDataDefault) pvm_pkint ( Sme , 1 , 1 ) ;

pvm_pkfloat (Sresult, 1, 1);

msgtype =5; master = pvm_parent ( ) ;

pvm_send (master, msgtype);

pvm_exit ( ) ;
```

Модификация виртуальной машины

Основные подпрограммы динамического изменения конфигурации добавляют в виртуальную машину новый хост и удаляют из нее старый:

```
int ierr = pvm_addhosts(char **hosts, int nhost, int *ierrs)

PVMFADDHOST(HOST, IERR)

int ierr = pvm_delhosts(char **hosts, int nhost, int *ierrs)

PVMFDELHOST(HOST, IERR)
```

Их параметры:

- hosts — массив указателей на строки символов, содержащие имена добавляемых/удаляемых машин;
- nhost — количество добавляемых/удаляемых хостов;
- ierrs — целочисленный массив из nhost элементов, содержащий коды завершения операции для каждого хоста. Отрицательные значения соответствуют ошибкам;
- host — строка символов, содержащая имя добавляемой/удаляемой машины (в FORTRAN).
- ierr — статус. Значения, меньшие nhost, указывают на частичную неудачу операции, а ierr, меньшее 1, означает полную неудачу операции.

Функции C позволяют сразу добавить или сразу удалить несколько хостов, а в программе на языке FORTRAN за одно обращение возможно добавление или удаление только одного хоста. При этом параметру ierr присваивается значение 1 или код ошибки. В C смысл параметра ierr иной — это количество хостов, которые действительно удалось включить в состав виртуальной машины.

С помощью подпрограмм pvm_addhosts и pvm_delhosts можно повысить устойчивость параллельной программы, а также увеличить эффективность использования вычислительных ресурсов. Если в процессе работы программе потребуются дополнительные ресурсы, она может самостоятельно изменить конфигурацию виртуальной машины в соответствии со своими потребностями, добавив в нее новые хосты. Ну а если потребность в ресурсах снизилась, ненужные хосты могут быть исключены из конфигурации. Если оказывается, что какой-то из хостов не удается включить в виртуальную машину, программа может подключить другой хост из числа доступных.

Если при работе хоста происходит сбой, PVM продолжает работать, но этот хост автоматически удаляется из виртуальной машины. Возможные ошибки:

PvmBadParam, PvmSysErr и PvmOutOfRes (нехватка ресурсов).

Подпрограмма pvm_hait действует аналогично команде halt PVM-кон-соли — она завершает работу виртуальной машины вместе со всеми задачами, которые, возможно, выполняются в момент вызова:

int ierr = pvm_halt(void) PVMFHALT(IERR)

Единственным параметром этой подпрограммы является код завершения ierr. Очевидно, pvm_hait следует использовать осторожно (да и стоит ли использовать?). При выполнении может возникнуть стандартная ошибка FvmSysErr — не отвечает локальный демон pvmd.

Подпрограммы pvm_setopt и pvm_getopt позволяют программисту задать режимы работы PVM или определить параметры текущего режима:

int oldval = pvm_setopt(int what, int val)

PVMFSETOPT(WHAT, VAL, OLDVAL)

int val = pvm_getopt(int what) PVMFGETOPT(WHAT, VAL)

Для указания режима используется параметр what (табл. 7.2), его значение — параметр val.

Таблица 7.2. *Допустимые значения параметра what подпрограммpvm_setopt И pvm_getopt*

Символьное значение	**Численное значение**	**Описание**
PvmRoute	1	Политика маршрутизации
PvmDebugMask	2	Маска отладки
PvmAutoErr	3	Автоматическое извещение об ошибках
PvmOutputTid	4	Переназначение стандартного вывода (stdout) дочерних процессов
PvmOutputCode	5	Тег сообщений с переназначенным стандартным выводом
PvmTraceTid	6	Переназначение вывода результатов трассировки дочерних процессов

PvmTraceCode	7	Тег сообщений с данными трассировки
PvmFragSize	8	Размер фрагментации сообщений
PvmResvTids	9	Резервировать теги и идентификаторы задач для сообщений
PvmSelfOutputTid	10	Переназначение стандартного вывода (stdout) для вызывающего процесса
PvmSelfOutputCode	11	Тег сообщений с переназначенным стандартным выводом для вызывающего процесса
PvmSelfTraceTid	12	Переназначение вывода результатов трассировки для вызывающего процесса
PvmSelfTraceCode	13	Тег сообщений с данными трассировки вызывающего процесса
PvmShowTids	14	Включать в перехваченный подпрограммой pvm_catchout вывод идентификаторы задач
PvmPollType	15	Политика ожидания сообщений

PvmPollTime	16	Продолжительность ожидания сообщений

Подпрограмму pvm_setopt часто используют для включения режима прямой маршрутизации (direct route communication). При вызове:

pvm_setopt(PvmRoute, PvmRouteDirect)

пропускная способность сети удваивается, однако при этом в ОС UNIX теряется масштабируемость программы, которая может оказаться неработоспособной при числе процессов более 60. В этом случае система автоматически сама переходит в режим по умолчанию.

Параметр PvmRoute определяет, какой режим маршрутизации следует использовать для последующих обменов между задачами. В PVM имеется набор предопределенных значений этого параметра (табл. 7.3).

Таблица 7.3. *Предопределенные значения параметра PvmRoute*

PvmDontRoute	1	Не запрашивать и не предоставлять связь
PvmAllowDirect	2	Не запрашивать, но предоставлять по запросу связь. Принято по умолчанию
PvmRouteDirect	3	Запрашивать и предоставлять по запросу связь

Параметр PvmRouteDirect (используется сетевой протокол TCP/IP) включает прямую связь между задачами для всех последующих коммуникаций. После установки линии связи она сохраняется до завершения работы программы. Если прямая связь не может быть установлена из-за того, что одна из задач запросила PvmDontRoute или оказались недоступными необходимые ресурсы, то будет использоваться маршрут передачи сообщений, принятый системой по умолчанию. В мультипроцессорных системах это значение игнорируется, а коммуникации между задачами построены на собственных протоколах передачи данных. Подпрограмму pvm_setopt можно вызывать неоднократно для избирательной установки прямых связей между задачами, однако обычно установка необходимых режимов выполняется один раз в начале работы

каждого процесса. Установкой по умолчанию является pvmAiiowoirect. Если данный режим установлен одной из задач, всем прочим задачам разрешается устанавливать с ней прямую связь.

Ключ FvmDebugMask включает режим отладки, когда информация об операциях протоколируется и направляется в стандартный поток stderr. Уровень отладки задается параметром vai. По умолчанию отладочная информация не выводится.

Если при вызове подпрограммы из библиотеки libpvm происходит ошибка, а значение PvmAutoErr равно 1 (это значение принято по умолчанию), сообщение об ошибке выводится в стандартный поток ошибок stderr. Нулевое значение отключает вывод. При значении 2 после вывода сообщения об ошибке выполнение задачи завершается системным вызовом exit. При значении параметра 3 после вывода сообщения об ошибке прерывается выполнение подпрограммы.

Вызов подпрограммы с ключом PvmOutputTid определяет, куда будет направляться стандартный вывод (stdout) для дочерних процессов, запущенных после вызова подпрограммы pvm_setopt. Все, что передается дочерними процессами в стандартный вывод, упаковывается в сообщения, которые посылаются по указанному адресу. В этом случае параметр vai является идентификатором процесса-адресата. Нулевое значение PvmOutputTid направляет стандартный вывод главному демону pvmd, который записывает протокол в файл /tmp/pvml.UID. Значение по умолчанию наследуется от родительской задачи, иначе это 0.

Ключ pvmOutputcode "используется для установки тега сообщений стандартного вывода совместно с ключом PvmOutputTid.

С помощью ключа pvmTraceTid можно определить, куда будут направляться данные трассировки для дочерних задач, запущенных после вызова подпрограммы pvm_setopt. Результаты трассировки передаются адресату в виде сообщений. В этом случае vai является идентификатором задачи-адресата. Нулевое значение отключает трассировку. Значение по умолчанию наследуется от родительской задачи или равно 0.

Ключ pvmTraceCode позволяет установить тег сообщений с данными трассировки, используется совместно с ключом PvmTraceTid.

Для ключа pvmFragSize значение vai определяет размер фрагмента сообщения в байтах. Значение по умолчанию зависит от архитектуры.

Для ключа pwiResvTids значение val = 1 позволяет задаче передавать сообщения с зарезервированными тегами и по адресам, не относящимся к задаче. Нулевое значение, принятое по умолчанию, заставляет систему возвращать сообщение об ошибке PvmBadParam, если задан зарезервированный идентификатор.

Ключ FvmSeifOutputTid определяет, куда следует направлять стандартный вывод задачи, из которой выполняется вызов подпрограммы. Все, что поступает в поток стандартного вывода, упаковывается в сообщения и передается по указанному адресу. Эта возможность работает только с задачами, запущенными из программы. Значением val является идентификатор задачи (TID). При нулевом значении ключа pvmSeifOutputTid стандартный вывод перенаправляется главному демону pvmd, который записывает его в файл /tmp/pvml.UID. Значение по умолчанию наследуется-от родительской задачи или равно 0. Установка PvmSeifOutputTid или pvmseifoutputcode приводит

К присваиванию PvmOutputTid И PvmOutputCode значений PvmSeifOutputTid и pvmseifoutputcode соответственно.

Ключ Pvmseifoutputcode позволяет задать тег для сообщения, содержащего данные из стандартного вывода.

С помощью pvmSelfTraceTid можно задать, куда будет направляться сообщение с данными трассировки задачи. Идентификатор задачи-адресата задается параметром val. Нулевое значение ключа PvmSelfTraceTid отменяет трассировку. Значение по умолчанию наследуется от родительской задачи либо равно нулю. Задание PvmSelfTracGTid или PvmSelfTraceCode приводит к присвоению PvmTraceTid и PvmTraceCode значений PvmSelfTraceTid и PvmSelfTraceCode соответственно.

Тег для сообщений с результатами трассировки задается с помощью ключа PvmSeIfTraceCode.

При ненулевом значении pvmshowTids подпрограмма pvm_catchout помечает каждую строку вывода дочерней задачи ее идентификатором. Для задания политики ожидания сообщений при их передаче через разделяемую память используется ключ pvmPoilType. Значение PvmPoilconstant вынуждает прикладную программу "прокручивать" очередь в ожидании сообщения. Значение pvmPoiisieep вынуждает программу опрашивать очередь о сообщениях PvmPollTime перед использованием семафора. Счетчик опроса устанавливается с помощью ключа PvmPoiiTime. Включить режим оповещения о наступлении особых событий можно с помощью подпрограммы pvm_notify:

int ierr = pvm_notify(int what, int msgtag, int cnt, int *tids) PVMFNOTIFY(WHAT, MSGTAG, CNT, TIDS, IERR)

Параметры у этой подпрограммы такие:

- what — тип уведомления *(см. гл. 6);*
- msgtag — тег сообщений, которые будут использоваться для уведомления;
- cnt — длина массива tids для уведомлений о событиях pvmTaskExit и

pvmHostDeiete, а для события PvmHostAdd задает количество уведомлений;
- tids — целочисленный массив длины ntask, содержащий список задач или демонов, о которых должна быть извещена задача, содержащая вызов подпрограммы pvm_notify. Для события PvmHostAdd этот массив должен быть пустым;
- ierr — код завершения. Отрицательное значение соответствует .ошибке.

В случае возникновения особых событий вызвавшая данную подпрограмму задача получает сообщения, формат которых зависит от типа события:

- PvmTaskExit — для каждого идентификатора tid передается одно сообщение, которое содержит идентификатор задачи, завершившей работу;
- PvmHostDeiete — для каждого идентификатора tid передается одно сообщение, содержащее идентификатор демона, завершившего работу;
- PvmHostAdd — в этом случае могут передаваться несколько сообщений, но не более cnt. Каждое сообщение содержит целочисленный счетчик, после которого следует список идентификаторов новых демонов. Нулевое значение cnt отключает сообщения, а — 1 — включает.

Идентификаторы задач в сообщениях упаковываются в целый формат. Вызывающая задача должна принять уведомление и обработать его самостоятельно. Возможные ошибки — PvmSysErr и PvmBadParam.

Группы процессов

Для работы с динамическими группами процессов используется отдельная библиотека libgpvmS, которую следует подключать на этапе сборки программы. Демон pvmd не умеет работать с группами, это может делать сервер группы, который автоматически запускается при первом вызове групповой подпрограммы (т. е. подпрограммы, выполняющей какую-либо групповую операцию).

Любая задача PVM может входить в любую группу и покидать ее в любой момент времени. Группа в PVM является динамическим объектом, ее состав может меняться в процессе выполнения программы. Процессы могут передавать сообщения групповой рассылкой тем группам, членами которых они не являются. Любая задача может вызвать любую групповую операцию, а подпрограммы pvm_lvgroup, pvm_barrier И pvm_reduce могут вызываться только из процесса, являющегося членом группы.

Управление группой

Подпрограммы pvm_joingroup и pvm_ivgroup позволяют задаче войти в указанную группу или покинуть группу соответственно:

int inum = pvm_joingroup(char *group)

PVMFJOINGROUP(GROUP, INUM)

int ierr = pvm_lvgroup(char *group)

PVMFLVGROUP(GROUP, IERR)

Первое обращение к pvm_joingroup создает группу с именем group и помещает процесс в эту группу. Подпрограмма pvm_j oingroup возвращает *групповой идентификатор* inum (от английских слов *instance number* — номер экземпляра) процесса в данной группе. В *главе 6* мы уже упоминали о том, что групповые идентификаторы изменяются от 0 до максимального значения, равного количеству процессов в группе минус один. Отрицательное значение говорит о возникновении ошибки. Одна задача может одновременно быть членом нескольких групп.

Если процесс вначале покидает группу, а затем вновь входит в нее, он получит, скорее всего, другой групповой идентификатор, со значением, наименьшим из доступных. Сохранение значения inum при повторном включении в группу не гарантируется.

Для того чтобы обеспечить непрерывность присвоения групповых идентификаторов независимо от изменения состава группы, подпрограмма pvm_ivgroup завершает свою работу только после того, как она получит подтверждение от задачи о выходе из группы. Если после этого вызывается подпрограмма pvmjoingroup, процессу будет назначено "свободное" значение группового идентификатора. При работе с группами процессов каждый процесс однозначно определяется парой (group, inum).

При выполнении подпрограмм pvm_joingroup и pvm_lvgroup возможны ошибки:

- FvmSysErr — не работает демон pvmd;
- pvmBadParam — задано имя пустой группы;
- PvitiNoGroup — указанной группы нет (для pvm_lvgroup);
- PvmDupGroup — попытка войти в группу, членом которой данный процесс уже является (для pvm_joingroup);
- pvmNotinGroup — попытка покинуть группу, членом которой данный процесс не Является (для pvm_lvgroup).

Пример использования подпрограмм pvm_j oingroup и pvm_lvgroup приведен в листингах 7.4 и 7.5. Исполняемый файл подчиненной программы должен быть сохранен с именем joinleave. Главная задача запускает 4 подчиненных

задачи, каждая из которых несколько раз входит в состав группы JoinLeave и выходит из нее. Разберите работу этого примера.

Листинг 7.4. Главная программа примера join leave

```
Mnclude <stdio.h>
#include "...pvm3/include/pvm3.h"
main()
{
int mytid;
int tids[32];
int n, nproc = 2, numt, i;
mytid = pvm_mytid () ;
{
for (i = 0; i <= 3; i++)
numt=pvm_spawn("joinleave", (char**)0, 0, "", nproc, tids);
pvm_exit();
}
}
```

Листинг 7.5. Подчиненная программа примера joinleave

```
tfinclude <stdio.h>
#include "...pvm3/include/pvm3.h"
int
main()
{
int mytid, mygid, i;
mytid = pvm_mytid();
if (mytid < 0) {
pvm_perror("Error!!!");
```

```
return -1;

}

ford = 1; i < 10; i++) {

if((mygid = pvm_joingroup("JoinLeave")) < 0)

pvm_perror("JoinLeave");

}

if((mygid = pvm_lvgroup("JoinLeave")) < 0) {

pvm_perror("JoinLeave") ;

}

}

}
```

Информационные подпрограммы:

```
int tid = pvm_gettid(char *group, int inum)

PVMFGETTID(GROUP, INUM, TID)

int inum = pvm_getinst(char *group, int tid)

PVMFGETINST(GROUP, TID, INUM)

int size = pvm_gsize(char *group)

PVMFGSIZE(GROUP, SIZE)
```

Подпрограмма pvm_gettid возвращает идентификатор tid процесса, принадлежащего к указанной группе (group — символьная строка, содержащая имя группы) и с указанным групповым идентификатором. Отрицательное значение tid возвращается при ошибке выполнения. Подпрограмма pvm_gettid позволяет двум процессам получить идентификатор.ы друг друга просто вхождением в общую группу. Подпрограмма pvm_getinst возвращает групповой идентификатор процесса с указанным TID в группе. Подпрограмма pvm_gsize возвращает количество процессов в группе.

Возможны следующие ошибки выполнения:

- FvmSysErr, pvmNoGroup — нет указанной группы;
- pvmNoInst — нет заданного группового идентификатора;
- PvmNotinGroup — указанный процесс не входит в заданную группу и некоторые другие.

Групповые рассылки сообщений и синхронизация процессов

Для выполнения групповой рассылки сообщения используется подпрограмма pvm_mcast:

int ierr = pvm_mcast(int *tids, int ntask, int msgtag)

PVMFMCAST(NTASK, TIDS, MSGTAG, IERR)

Параметры:

- ntask -- количество задач, которым должно передаваться содержимое активного буфера;
- tids — целочисленный массив из ntask элементов, который содержит идентификаторы задач-адресатов;
- msgtag — тег сообщения;
- ierr — код завершения.

Вызывающей подпрограмме сообщение не пересылается в любом случае, даже если по какой-то причине ее идентификатор указан в массиве tids. Прием сообщения, отправленного групповой рассылкой, может выполняться подпрограммами pvm_recv И pvm_nrecv. Операция передачи pvm_mcast асинхронна (неблокирующая), т. е. после передачи сообщения в коммуникационную сеть задача может продолжать работу.

Групповую рассылку не поддерживают многие производители мультипроцессорных систем, поддерживая лишь рассылку всем процессам. В этом случае следует познакомиться с особенностями работы PVM на данной мультипроцессорной системе.

Возможные ошибки при выполнении групповой рассылки: *П* FvmBadParam — отрицательный тег сообщения; П pvmSysErr — не отвечает локальный демон; *П* pvmNoBuf — отсутствует указанный буфер.

Кроме групповой, в PVM реализована и широковещательная рассылка, которая выполняется подпрограммой pvm_bcast:

int ierr = pvm_bcast(char *group, int msgtag)

PVMFBCAST(GROUP, MSGTAG, IERR)

Эта подпрограмма присваивает сообщению идентификатор msgtag и выполняет его

широковещательную рассылку *всем* задачам из группы (в этом ее отличие от рассылки групповой), которая задается именем (параметр group — символьная строка). Если процесс входит в группу уже после того, как рассылка началась, он может и не получить сообщение. Если же во время рассылки процесс покидает группу, сообщение он все равно получит. Процесс, выполняющий широковещательную рассылку, не обязательно должен быть членом группы. Широковещательная рассылка является неблокирующей (асинхронной) операцией. Перед выполнением рассылки подпрограмма pvm_bcast сначала определяет идентификаторы членов группы по базе данных групп, а затем передает содержимое активного буфера соответствующим процессам. При выполнении данного вызова возможны ошибки PvmSysErr, pvmBadParam и PvmNoGroup (указанная группа не существует).

Подпрограмма pvm_reduce:

int ierr = pvm_reduce(void (*func)(), void *data, int nitem,

int datatype, int msgtag, char *g'roup, int root)

PVMFREDUCE(FUNC, DATA, COUNT, DATATYPE, MSGTAG, GROUP, ROOT, IERR)

выполняет операции приведения в группе. Результат операции пересылается главному процессу. В PVM есть четыре предопределенных операции:

PvmMax PvmMin PvmSum PvmProduct

Их смысл понятен из названий — вычисление максимума и минимума, суммы и произведения. Операция приведения выполняется поэлементно над входными данными так же, как это делается в MPI.

Параметры подпрограммы pvm_reduce:

- func — операция приведения;
- data — указатель на начальный адрес массива локальных значений. После выполнения подпрограммы массив значений в главной задаче модифицируется и заменяется результатом выполнения операции приведения;
- count — количество элементов в массиве данных;
- datatype — тип массива данных;
- msgtag — тег сообщения;
- group — имя группы (символьная строка);
- root — групповой идентификатор главного процесса;
- ierr — код завершения подпрограммы.

Могут быть определены пользовательские операции приведения. Интерфейс пользовательской операции имеет вид в C:

void func(int *datatype, void *x, void *y, int *num, int *ierr) и в FORTRAN:

CALL FUNC(DATATYPE, X, Y, NUM, IERR)

Параметры x и y являются массивами, их тип задается параметром datatype, а количество элементов — параметром num. Параметр x соответствует параметру data подпрограммы pvm_reduce, а y содержит принимаемые данные.

Операция pvm_reduce является неблокирующей. Если процесс обращается к pvm_reduce, а потом покидает группу до того, как главный процесс обратится к pvm_reduce, возникнет ошибка.

Операции PvmMax и PvmMin выполняются для типов byte, short, integer, long, float, double, complex и double complex. В случае комплексных параметров максимальное или минимальное значение определяется для модуля. Операции pvmsum и pvmProduct определены для операндов, имеющих тип short,

integer, long, float, double, complex И double complex. Соответствие типов в языках C и FORTRAN приведено в табл. 7.4.

Таблица 7.4. *Соответствие типов в языках C и FORTRAN*

Тип в C	**Тип в FORTRAN**
PVM_BYTE	BYTE1
PVM SHORT	INT2
PVM_INT	INT 4
PVM FLOAT	REAL4
PVM_CPLX	COMPLEX8
PVM DOUBLE	REALS
PVM DCPLX	COMPLEX16

PVM LONG	

Вероятные ошибки выполнения подпрограммы pvm_reduce: PvmBadParam (неправильное значение аргумента), pvmNoinst (вызывающий процесс не входит в группу) и PvmSysErr (не отвечает демон pvmd).

Подпрограмма pvm_barrier используется для синхронизации процессов и действует так же, как аналогичная подпрограмма в MPI:

int ierr = pvm_barrier(char *group, int count)

PVMFBARRIER(GROUP, COUNT, IERR)

Она блокирует работу процесса до тех пор, пока к ней не обратятся count членов группы. Обычно значение параметра count совпадает с количеством процессов в группе. Наличие данного параметра объясняется тем, что уже после вызова pvm_barrier состав группы может измениться, какие-то процессы могут выйти из группы, а какие-то войти в нее. Не допускается обращение к подпрограмме pvm_barrier из процесса, не принадлежащего группе. Кроме того, значения count при всех обращениях должны совпадать. Группа задается ее именем — символьной строкой group. Код завершения содержится в переменной ierr. Отрицательное значение возвращается при возникновении ошибки. Если значение count = -l, для определения количества процессов в группе PVM использует функцию pvm_gsize. Это значение рекомендуется применять, если состав группы не изменяется.

При выполнении этой подпрограммы возможны ошибки: PvmSysErr, pvmBadParam (если count < l), PvmNoGroup (если нет указанной группы) и pvmNotInGroup (вызывающий процесс не является членом указанной группы).

Примеры использования групповых операций приведены в листингах 7.6 и 7.7. Здесь имеются операции широковещательной рассылки, выполняется синхронизация процессов, а также собирается информация о группе. Предлагаем читателю самостоятельно разобраться в деталях работы этих программ и попробовать запустить их на выполнение.

Листинг 7.6. Пример использования групповых операций

```
#include <stdio.h>

#include ". . .pvm3/include/pvm3 . h"

main ( )
```

```
{
int mytid, mygid, ctid [32];
int nproc = 2, i, indx;
int cc;
mytid = pvm_mytid ( ) ;
mygid = pvm_joingroup ("ge") ;
if (mygid ==0) (
pvm_spawn("ge", (char**) 0, 0, "", nproc — 1, ctid);
pvm_initsend(PvmDataDefault) ;
pvmjpkint (&rip roc, 1, 1) ;
pvm_mcast (ctid, nproc — 1, 15);
}
else (
pvm_recv (pvm_parent ( ) , 15 );
pvm_upkint ( Snproc , 1 , 1 ) ;
}
pvm_barrier ("ge", nproc);
pvm_initsend(PvmDataDefault) ;
pvm_pkint ( Smygid, 1, 1) ;
pwypkint (Smytid, 1, 1);
pvm_bcast ("ge", 63);
for (i = 0; i < nproc — 1; i++)
{ pvm_recv(-l, 63); pvm__upkint ( & indx , 1 , 1 ) ;
pvm_upkint (ctid + indx, 1, 1) ;
```

```
}

pvm_lvgroup ( "ge" ) ; pvm exit ( ) ; return 0;

}
```

Листинг 7.7. Пример определения размера группы

```
#include <stdio.h>

#include "...pvm3/include/pvm3.h"

#define GSGROUP "gsgroup"
main()
{
int mytid, mygid, ctid[32];
int nproc =2, i;
mytid = pvm_mytid();
mygid = pvm_joingroup(GSGROUP);
if (mygid ==0) {
pvm_spawn("gs", PvmTaskDefault, 0, "", nproc — 1, ctid);
}
whiJ,e (pvm_gsize (GSGROUP) < nproc)
if (mygid == 0)
fputs("waiting on children processes to join\n", stderr);
pvm_barrier(GSGROUP, nproc); pvm_lvgroup(GSGROUP);

pvm_exit (); return 0;
```

Вопросы и задания для самостоятельной работы

1. Проведите исследование быстродействия глобальных операций PVM для разного числа процессов и разных размеров сообщений.
2. Приведите пример приложения, в котором необходимо использовать, по крайней мере, два буфера.
3. Выполните задания 4—9 главы 5, используя PVM. Сравните полученный опыт с опытом решения тех же самых задач с применением MPICH. Сделайте выводы.

ГЛАВА 8.

Высокопроизводительный FORTRAN

High Performance FORTRAN (HPF) или Высокопроизводительный FORTRAN

представляет собой расширение языка программирования FORTRAN-90 и достаточно широко применяется для разработки параллельных программ. Данная глава содержит необходимые сведения о языке программирования FORTRAN-90 и обзор основных директив HPF. Приводятся примеры использования конструкций HPF, а в заключение предлагаются задания для самостоятельной работы.

Общие сведения о HPF

Команды HPF сводятся к небольшому числу директив, реализующих парадигму параллелизма данных и SPMD-модель программирования. В модели программирования, которую поддерживает HPF, все процессоры выполняют одну и ту же программу. Каждый процесс оперирует с отведенной ему частью общего массива данных, данные распределяются по процессорам директивами HPF, а параллелизм определяется инструкциями FORTRAN-90 и HPF.

Создатели HPF постарались избавить прикладного программиста от явной передачи сообщений, сравнив модель передачи сообщений с программированием на языке ассемблера. Действительно, программам, написанным на HPF, никогда не достигнуть эффективности их эквивалентов, реализованных средствами передачи сообщений. Однако в некоторых случаях время, сэкономленное на написании и отладке программы на HPF, с лихвой покрывает выигрыш в производительности аналогичной программы, написанной с применением MPI.

Кроме того, программа, написанная на языке FORTRAN-90, без всяких изменений пропущенная через компилятор HPF, будет распараллелена только с использованием системных средств, и даже такой примитивный способ распараллеливания может привести к значительному приросту в скорости исполнения.

HPF был разработан группой HPF Forum, главную роль в которой играли представители крупных компаний, выпускающих компьютеры: Convex, Cray Research, DEC, Fujitsu, Hewlett Packard, IBM, Intel, Meiko, Sun Microsystems, Thinking Machines. Группа впервые собралась на конференции Supercompu-ting'91 с намерениями быстро получить результат. Предварительные соглашения появились уже к следующей конференции (Supercomputing'92), а первая версия стандарта (HPF v1.0) была распространена в мае 1993 г. Вторая версия HPF v2.0 вышла в 1996 г.

Если MPICH и PVM являются свободно распространяемыми программными продуктами, большая часть компиляторов HPF — коммерческие. В Интернете можно найти бесплатные демонстрационные версии компиляторов с ограниченными возможностями, но наиболее популярным является пакет PGI Server фирмы Portland Group Compiler Technology. Пакет распространяется с пробной 15-дневной лицензией, которая может реконструироваться в течение года. В состав пакета входят компиляторы с языков C/C++, FORTRAN-77, FORTRAN-90 и HPF с утилитами отладки и профилирования.

Новое в FORTRAN-90

FORTRAN-90 продолжает оставаться строго лицензионным продуктом, что препятствует его широкому распространению. Изменения, внесенные именно в эту редакцию языка FORTRAN, подняли его на качественно новый уровень и существенно расширили возможности программирования, сохранив основные достоинства языка — простоту и естественность. Знаний только предыдущих версий: FORTRAN-4, 66 и 77 недостаточно для эффективного использования конструкций FORTRAN-90 и, следовательно, HPF. Мы не ставим задачу привести здесь сколько-нибудь полное описание FORTRAN-90. Скорее, это обзор языка с целью обратить внимание читателя на те инструкции, которые существенны для программирования на HPF. В первую очередь это черты, отсутствующие в предыдущих стандартах языка FORTRAN и имеющие принципиальное значение для эффективного использования FORTRAN-90 в качестве базового языка HPF.

Программные единицы и структура программы

Структура программы, принятая в FORTRAN-90, обладает гибкостью, логической строгостью и полнотой. Следование правилам структурирования в сильной степени способствует написанию хороших программ. Помимо основной программы и подпрограмм, принятых в ранних версиях стандарта, введены принципиально новые для данного языка программные единицы -модули. Внесено разделение подпрограмм на внутренние и внешние, добавлены интерфейсы. Использование этих нововведений позволяет отслеживать на этапе компиляции такие традиционные для FORTRAN ошибки, как несоответствие типов формальных и фактических параметров процедур.

Лексические единицы и запись исходного текста

Исходный текст программы формируется из основного набора символов, который содержит буквы A—Z, как прописные, так и строчные (регистр не имеет значения), знак подчеркивания _, цифры 0—9 и специальные символы:

,:;<>?%"&$.!=+-*/() [ ] CR TAB Пробел

Основные значимые последовательности из алфавитно-цифровых или специальных символов называются *лексемами.* К ним относятся метки, ключевые слова, имена, константы, знаки операций и разделители:

/()(//),==>:::;%

Разделителями лексем являются пробелы. Пробелы, не искажающие смысла лексем, допускаются в любом количестве, а последовательность из нескольких пробелов равносильна одному пробелу. Операторы разделяются либо точкой с запятой (;), либо символом новой строки. Каждый законченный оператор может быть помечен. Метка в

виде целого числа, содержащего от одной до пяти цифр, из которых хотя бы одна отлична от нуля, помещается перед помечаемым оператором.

Кроме традиционного *фиксированного формата* записи исходного текста, вводится так называемый *свободный формат.* В этом формате допускаются строки длиной до 132 символов, а позиция символа в строке не имеет значения. Начало комментария отмечается символом ! и может располагаться в любой позиции; концом комментария является конец строки. Символ &, помещенный в конце строки, отмечает незаконченную строку: следующая строка будет рассматриваться компилятором как продолжение начатого текста. Если в строке продолжения присутствует знак &, то продолжением считается та часть строки, которая следует за ним. Компилятор (по крайней мере, PGI) по умолчанию готов к принятию традиционно форматированного текста (первые пять позиций в строке для метки и шестая для признака продолжения строки). Для восприятия свободно форматированного исходного текста компилятору требуется ключ — Mfreeform.

Главная программа и внешние подпрограммы

Обязательным компонентом программ является главная программа — точка входа, которой системное окружение передает управление при вызове исполняемого файла программы. Любая программа обязана содержать одну и только одну главную программу. Главная программа представляет собой последовательность операторов, завершающуюся единственным обязательным ключевым словом END. Необязательное ключевое слово PROGRAM может начинать текст главной программы, кроме того, главная программа может иметь необязательное имя, помещаемое после слова PROGRAM в начале и конце текста программы. Общая структура главной программы такова:

[PROGRAM program_name]

[операторы описания]

[исполняемые операторы]

[CONTAINS

описания внутренних подпрограмм]

END [PROGRAM [program_name]]

Квадратные скобки здесь и в дальнейшем отмечают необязательные элементы. Регистр символов не имеет значения (если нет специального указания компилятору), но для определенности мы будем ключевые слова воспроизводить прописными буквами, а имена элементов — строчными.

Оператор CONTAINS отмечает начало описания внутренних подпрограмм. При выполнении программы он осуществляет передачу управления оператору END, при наличии внутренних подпрограмм его присутствие обязательно. Ключевое слово END в главной программе означает безусловный останов программы и возврат управления операционной системе. Хорошо организованная программа имеет единственную точку выхода — оператор END главной программы, хотя это вовсе не обязательно. Оператор STOP, останавливающий работу программы, может быть помещен в любой подпрограмме.

Внешняя подпрограмма содержит последовательность операторов, вызываемую из другой программной единицы. Структура внешней подпрограммы похожа на структуру главной подпрограммы, но есть и отличие — ее начало отмечается обязательным заголовком подпрограммы, состоящим из ключевого слова, обязательного имени подпрограммы и списка формальных параметров, если они есть. Ключевое слово зависит от вида подпрограммы: это SUBROUTINE для подпрограмм-процедур и FUNCTION для подпрограмм-функций. В заголовке функций может указываться тип возвращаемого значения. Описание внешней подпрограммы заканчивается обязательным ключевым словом END, за которым может следовать слово SUBROUTINE (или соответственно FUNCTION) и имя подпрограммы:

[тип_возвращаемого_значения] FUNCTION &

function_name[(список_формальных_параметров)]

[операторы описания]

[исполняемые операторы]

[CONTAINS

описания внутренних подпрограмм]

END [FUNCTION [function name]]

SUBROUTINE subroutine_name[(список_формальных_параметров)]

[операторы описания]

[исполняемые операторы]

[CONTAINS

описания внутренних подпрограмм]

END [SUBROUTINE [subroutine_name]]

Ключевое слово END, завершающее внешнюю подпрограмму, означает возврат управления в место вызова подпрограммы. Оператор RETURN используется для возврата управления в вызывающий блок из других мест подпрограммы.

Внутренние подпрограммы

Внутренняя подпрограмма является частью того программного компонента, в котором она размещена. Если внешняя подпрограмма выделяет некоторую подзадачу в рамках всей программы, внутренняя подпрограмма выделяет часть задачи в поле зрения включающего ее компонента-носителя и доступна только из него. Внутренняя подпрограмма автоматически имеет доступ ко всем переменным и подпрограммам элемента носителя. В случае совпадения имен локальных переменных внутренней подпрограммы с именами переменных компонента-носителя используются локальные переменные внутренней подпрограммы. Внутренняя подпрограмма не должна содержать подпрограмм — допустимый уровень вложенности ограничен. Общий вид внутренних подпрограмм:

SUBROUTINE имя_подпрограммы[(список_формальных_параметров)] [операторы описания] [исполняемые операторы] END SUBROUTINE [имя_подпрограммы]

или:

Тип_возвращаемого_значения] FUNCTION имя_функции &
[(список_формальных_параметров)] [операторы описания] [исполняемые операторы] END FUNCTION [имя_программы]

Интерфейсы

Традиционное для языка FORTRAN раздельное транслирование компонентов (блоков программы) приводит к переносу ошибок, связанных с несоответствием формальных и фактических параметров, в категорию ошибок времени исполнения, которые чрезвычайно трудно определить и локализовать. Компилятор способен проверить соответствие формальных и фактиче-

ских параметров только для внутренних подпрограмм. В этом смысле внутренние подпрограммы обладают *явным интерфейсом,* а внешние подпрограммы имеют лишь *неявный интерфейс.* В языке FORTRAN-90 введены новые программные единицы — *интерфейсы,* позволяющие объявлять заголовки и описания параметров внешних подпрограмм в вызывающих их программных компонентах. Интерфейсный блок имеет вид:

INTERFACE тело_интерфейса END INTERFACE

Тело интерфейса содержит точную копию заголовка подпрограммы, описание

формальных параметров и оператор END подпрограммы. В интерфейсном блоке можно изменять имена параметров и добавлять описания локальных переменных. Интерфейсные блоки должны размещаться среди операторов описания. Самым подходящим местом для интерфейсных блоков являются модули. Появление имени подпрограммы внутри интерфейсного блока (как и внутри оператора EXTERNAL) делает локальную функцию с таким же именем недоступной.

С помощью интерфейсов можно организовать *перегрузку процедур,* объединив под одним общим именем интерфейсы нескольких процедур. В этом случае интерфейс должен иметь имя, которое называется *родовым именем.* При вызове процедуры по родовому имени вызывается та процедура, формальные параметры которой совпадают с фактическими параметрами вызова. Необходимая процедура находится по типу и числу параметров, а не по имени. Процедуры, входящие в родовой интерфейс, должны иметь различия в формальных параметрах.

Модули и модульные процедуры

Модули используются для описания глобальных переменных и функций программы. Модули могут содержать описания переменных, namelist-группы, описания производных типов, интерфейсы внешних подпрограмм и описания собственных модульных процедур. Модули могут включать другие модули, но не могут обращаться сами к себе ни прямо, ни косвенно через цепочку включений. Общий вид модуля:

MODULE имя_модуля

[операторы описания]

[CONTAINS

модульные_подпрограммы]

END [MODULE [имя_модуля]]

Доступ к модулю осуществляется посредством оператора USE:

USE имя модуля

Модульные подпрограммы в своих операторах END обязаны содержать слова FUNCTION или, соответственно, SUBROUTINE и, как следствие, свои имена. В остальном они представляют собой обычные внешние подпрограммы. Объекты модуля, объявленные с модификатором доступа PRIVATE, недоступны из других программных компонентов. Модули являются развитием включений частей программного текста операторами INCLUDE и обобщением глобальных областей памяти, существовавших в виде comon-блоков. Использование модулей и интерфейсов для контроля глобальных переменных и функций очень полезно при

создании больших программных комплексов.

Рекурсивные процедуры и функции

Обычные процедуры и функции в языке FORTRAN не допускают обращения к самим себе. В FORTRAN-90 введены *рекурсивные* процедуры и функции. Заголовки рекурсивных подпрограмм должны содержать ключевое слово RECURSIVE. Так как рекурсивная функция не может сама содержать возвращаемое значение, она должна указывать имя переменной, в которой будет храниться результат. Для этого в ее заголовок вносится предложение RESULT:

RECURSIVE REAL FUNCTION имя_функции &

RESULT(имя_переменной_результата)

Если предложение RESULT отсутствует, имя функции используется для адресации результата и недоступно для рекурсивного вызова.

Формальные параметры подпрограмм

Ограничения на тип значений, передаваемых через параметры подпрограмм, отсутствуют. Наравне с переменными и массивами переменных числового и символьного типов можно передавать указатели и имена внешних подпрограмм. Для управления использованием параметров подпрограмм введен атрибут назначения параметров INTENT, который может принимать значения IN, OUT и INOUT. Атрибут INTENT применим только к тем параметрам, для которых он имеет смысл, параметры-указатели, например, не могут иметь атрибута INTENT. Параметры, помеченные как входные -INTENT (IN), не должны изменяться внутри подпрограммы. Выходные параметры (INTENT(OUT)) теряют свои значения при входе в подпрограмму и, если в подпрограмме они не получают новых значений, по выходе из нее оказываются в состоянии неопределенности. Параметры, имеющие статус :NOUT, обязаны быть переменными.

Допускаются *необязательные параметры* — они в описании подпрограммы помечаются ключевым словом OPTIONAL. Необязательные параметры могут опускаться при вызовах подпрограмм. Тогда в списке фактических параметров сначала в порядке следования перечисляются позиционные параметры,

а когда соблюдение порядка становится невозможным, оставшиеся параметры вводятся с ключами, соответствующими именам формальных параметров подпрограммы. Так, если в подпрограмме GetBounds из трех формальных параметров два являются необязательными, то при ее вызове с обязательным параметром size и необязательным form последний необходимо вводить с именем формального параметра:

SUBROUTINE GetBounds(size, length, form)

INTEGER size

INTEGER, OPTIONAL :: length, form

END SUBROUTINE GetBounds

CALL GetBounds(my_size, form = myform)

Если этого не сделать, значение my_form получит параметр length. Справочная функция PRESENT (имя__необязательного_формального_параметра) позволяет

проверить подстановку фактического параметра на место указанного необязательного формального.

Типы данных и операторы описания

Операторы описания

Принят новый синтаксис операторов описания, разделяющий указания типа и перечень атрибутов от списка переменных двойным двоеточием, например:

INTEGER, DIMENSION(10, 10) :: A,B,C

Неявное определение типа, характерное для всех версий FORTRAN, рекомендуется отключать, помещая первым оператором описания инструкцию:

IMPLICIT NONE

Базовые типы и их разновидности

Список основных типов, допускаемых в языке FORTRAN, сохранен. Это типы LOGICAL, CHARACTER, INTEGER, REAL И COMPLEX. Но в отношении разновидностей типов в смысле их точности внесены изменения, способствующие систематизации и унификации использования переменных различной точности. В операторы описания основных типов дан-ных входит необязательный параметр KIND, который указывает значение разновидности типа, вообще говоря, зависящее от аппаратной платформы используемого компьютера, но иногда совпадающее с числом битов, отводимых под переменную. Так, для описания вещественной переменной двойной точности, наряду

Назначение ссылки выполняется оператором =>:

переменная_указатель => переменная_адресат

Освобождение ссылки (разрыв связи указатель-переменная) производится оператором NULLIFY:

NULLIFY(переменная_указатель)

Указатели

Переменная-указатель может находиться в одном из трех состояний: привязанности, непривязанности и в состоянии неопределенности. Справочная встроенная функция:

ASSOCIATED(POINTER, [TARGET])

возвращает значение .TRUE., если ссылка POINTER находится в состоянии привязанности, и .FALSE, -в противном случае. Если POINTER находится в состоянии неопределенности, применение функции ASSOCIATED недопустимо.

Производные типы

В FORTRAN-90 есть возможность создавать собственные типы данных, комбинируя их из базовых. Фактически, это структуры, широко применяемые в языке С. Описание типа открывается оператором TYPE:

TYPE имя_производного_типа

описания компонентов END TYPE имя_производного_типа

Переменные созданного производного типа описываются следующим образом:

TYPE(имя_производного_типа) имя_переменной_производного_типа

Обращение к компонентам переменной производного типа выполняется с помощью оператора %:

TYPE COLORED_CIRCLE

REAL CX, CY, RADIUS

CHARACTER(LEN = 5) COLOR END TYPE COLORED_CIRCLE

TYPE(COLORED_CIRCLE) big_red, little_green, middle_pink big_red%COLOR = 'RED'; little_green%RADIUS =0.75

Переменные производного типа невозможно инициализировать в рамках операторов описания. Заполняется переменная при помощи *конструктора производного типа:*

имя типа(список_выражений)

и операции присваивания, например:

middle_pink = COLORED_CIRCLE(-2., -*1.,* 3., 'pink')

Операция присваивания переменной производного типа структуры, созданной конструктором, задается автоматически. Никакие другие операции для производных типов не определены. Если есть потребность в применении к производному типу операций, необходимо описать процедуру, производящую действия, ассоциируемые с операцией, и интерфейсный блок, связывающий знак операции с процедурой. После этого знак операции, применяемый к переменным данного производного типа, будет трактоваться как вызов соответствующей процедуры. Так для производного типа COLORED_CIRCLE можно описать процедуру объединения двух представителей этого типа UNITE и перегрузить оператор +:

INTERFACE OPERATOR(+)

MODULE PROCEDURE UNITE END INTERFACE

Тогда при "сложении" переменных типа COLORED_CIRCLE будет вызываться процедура UNITE. Можно также переопределить операцию присваивания для производного типа, когда переменной производного типа присваивается значение другого (производного или базового) типа, или наоборот. Естественно, за перегрузкой оператора присваивания должно стоять определение соответствующего действия — описание указанной в интерфейсе процедуры:

INTERFACE ASSIGNMENT(=)

MODULE PROCEDURE COLORED_TO_REAL, REAL_TO_COLORED

END INTERFACE

Описания производного типа и процедур, перегружающих операторы, должны помещаться в модулях, включаемых инструкцией USE во все компоненты программы, обращающиеся к данному производному типу.

Массивы в FORTRAN-90

Одним из основных достижений разработчиков стандарта FORTRAN-90 является возможность обращаться с массивами почти так же, как со скалярными переменными. Над массивами можно выполнять арифметические действия, их можно использовать в качестве аргументов встроенных функций, определен даже оператор управления, напоминающий условный оператор IF, применяемый к массивам. Другим большим достижением, несомненно, является введение динамических массивов, не нуждающихся в определении границ на стадии компиляции. Для многих действий над элементами массивов не требуется явного перебора элементов в цикле. Выборки

элементов массивов, называемые *сечениями,* также могут использоваться как массивы.

Строение массивов

В описаниях действий над массивами используются термины, характеризующие устройство массива: размер, ранг, экстент, форма.

Массивы FORTRAN-90 могут иметь до семи измерений — рангов. Число элементов в данном измерении называется *экстентом* массива в этом измерении. Целочисленный массив экстентов по всем измерениям массива есть *форма* массива. Общее число элементов массива называется *размером* массива. При описании массива фиксированного размера необходимо указывать верхние границы в измерениях, но можно определять и нижние границы: по умолчанию нумерация элементов начинается с 1, но это не обязательно. Пример:

REAL, DIMENSION(IO) :: A(-5, 4), B(21:31)

Следующие встроенные функции используются для получения информации о массиве:

- LBOUND(A) и UBOUND(A) - указывают значения нижней и верхней границ массива;
- SIZE (A) — сообщает размер массива;
- SHAPE (A) -- возвращает массив целых чисел, описывающих форму заданного массива.

Сечения массивов

В языке FORTRAN-90 возможно обращение к подмножеству элементов массива. Подмножество, называемое *сечением массива,* описывается диапазоном индексов (вместо набора индексов), по которому осуществляется обращение к элементу массива. Сечение массива само является массивом, за исключением того, что обозначением сечения нельзя пользоваться для ссылок на его отдельные элементы.

Можно, например, создать ссылку на подмножество элементов массива A:

REAL, TARGET, DIMENSIONS, 100) ::

A REAL, POINTER :: P

p => A(2, 1:100:2)

и обращаться в дальнейшем с переменной p как с массивом ранга 1 и размерности 50, например:

P(3)

Нельзя обращаться к элементам сечения, непосредственно указывая индекс элемента в сечении:

A(2, 1:100:2)(3)

Выборка индексов для сечения может быть задана целочисленным массивом *(векторным индексом).*

INTEGER, DIMENSION(1O) ::

IPOINT(/1, 3, 6, 7, 11, 15, 18, 21, 24, 29/) REAL, DIMENSION(100) :: A

Далее можно обращаться к подмножеству массива A(IPOINT) .

Конструкторы массивов

Выражение

IPOINT(/1, 3, б, 7, 11, 15, 18, -21, 24, 29/)

задает одномерный массив-константу с помощью конструктора — списка элементов, заключенного в (/ /). Выражение, заключаемое в такие ограничители, может содержать арифметические операции и неявные циклы, например:

(/ (I * 0.1, I = 11, 15) /)

задает вещественный массив из пяти элементов (Ал, 1.2, 1.3, 1.4, 1.5/). Конструкторы допускаются только для одномерных массивов, однако одномерный массив можно переформировать с помощью функции RESHAPE в массив требуемой формы.

Динамические массивы

Память для массивов, размеры которых становятся известны только в процессе выполнения программы, может быть выделена динамически. Массив, получающий динамическую память, не может быть формальным параметром или результатом, возвращаемым функцией. Выделяемый массив при описании должен получить атрибут ALLOCATABLE:

REAL, DIMENSION(:, :) : : A

Границы выделяемого массива должны быть определены к моменту вызова оператора ALLOCATE, при описании указывается только ранг массива. Границы выделяемого массива должны быть целочисленным выражением, например:

ALLOCATE(A(2 * N, -10:N + 1))

Справочная функция ALLOCATED (имя_выделяемого_объекта) возвращает значение .TRUE., если объект находится в состоянии выделенное™, и .FALSE. -в противном случае. Возможно и третье состояние — состояние неопределенности, возникающее, в частности, тогда, когда происходит возврат из подпрограммы, где была выделена память под локальный для подпрограммы массив. По завершении работы с выделяемым массивом необходимо явно освободить память, занятую массивом:

DEALLOCATE (A)

Операции выделения и освобождения памяти могут закончиться неудачей. Необязательный выходной параметр STAT задает переменную целого типа, в которую записывается код завершения операции (0 в случае успеха и положительное значение в случае неудачи). При отсутствии параметра STAT неуспех операции ALLOCATE или DEALLOCATE приводит к останову программы. Выделение памяти может производиться и для ссылок. Освобождение (deallocation) адресата прикрепленной ссылки приводит ссылку в состояние неопределенности .

Инструкция *WHERE*

Инструкция WHERE выполняет в отношении массивов роль условного оператора. WHERE не адресуется к элементам массива по индексам, а применяется к массиву в целом:

WHERE (логическое_выражение-массив) присваивание_массива

Так, например, все отрицательные элементы массива A изменят значения на противоположные:

WHERE (A < 0) A = -A

Логическое выражение-массив формирует из исходного массива *массив-маску* (массив логического типа, конформный исходному, с элементами, имеющими значения .TRUE, или .FALSE.). Изменения, происходящие с самим массивом, не изменяют массив-маску.

Полная форма конструкции WHERE такова:

WHERE (логическое_выражение-массив)

присваивания_массивов ELSEWHERE

присваивания_массивов END WHERE

Внутри конструкции WHERE операторы перехода не допускаются. Если к присваиванию массивов привлекается элементная функция, она будет применена только к тем элементам, которые удовлетворяют заданному условию. Маскирование не распространяется на массивы, являющиеся аргументами неэлементных функций. Если конструкция:

WHERE (B > 0) B= 1.0 / LOG(B)

не приведет к ошибочному вычислению логарифмов от отрицательных элементов массива в, то в операторе:

WHERE (B > 0) B= B / SUM(LOG(B))

неэлементная функция зим попытается просуммировать логарифмы от всех элементов массива.

Встроенные функции для работы с массивами

Мы сосредоточимся на встроенных функциях, связанных с массивами, опуская другие (может быть, очень полезные) функции. Встроенные функции, связанные с массивами, можно разделить на 5 категорий:

- элементные функции;
- справочные функции;
- функции для создания и преобразования массивов;
- функции для редукции массивов;
- функции для операций над векторами и матрицами.

Элементные функции определены изначально для скалярных аргументов, но их можно применять и к совместимым массивам. Каждый элемент массива-результата получается таким, как если бы функцию применили к каждому элементу массива-аргумента. Таким образом работают все традиционные встроенные элементарные математические функции языка FORTRAN:

ACOS(x), ASIN(X), ATAN(X), ATAN2(Y,X), COS(X),

COSH(X), EXP(X), LOG(X), LOG10(X), SIN(X),

SINH(X), SQRT(X), TAN(X), TANH (X) ;

функции преобразования типов:

ABS(A), AIMAG(Z), AINT(A[, KIND]), ANINT(A[, KIND]),

CEILING(A), CMPLX(X[, Y] [, KIND]), FLOOR(A),

INT(A[, KIND]), NINT(A[, KIND]), REAL(A[, KIND]);

и элементные функции, не преобразующие тип:

CONJG(Z), DIM(X, Y), MAX(A1, A2[, A3, ...]),

MOD(A, P) , MODULO(A, P), SIGN(A, B)

Справочные функции для массивов предоставляют информацию о текущем состоянии массива:

- LOGICAL ALLOCATED (ARRAY) — определяет состояние выделяемого массива;
- INTEGER LBOUND (ARRAY [, DIM]) — возвращает значения нижних границ массива по всем измерениям, если аргумент DIM не задан, и нижнюю границу в измерении DIM — в противном случае;
- INTEGER UBOUND(ARRAY[, DIM]) — функция, аналогичная предыдущей. Определяет верхние границы массива;
- INTEGER SHAPE (SOURCE) - возвращает одномерный массив стандартных целых чисел, содержащий форму массива SOURCE. Может применяться к скалярам, тогда в результате возвращается нуль;
- INTEGER SIZE (ARRAY [, DIM]) — возвращает стандартное целое число, равное размеру массива ARRAY, если аргумент DIM присутствует, возвращается экстент по размерности DIM.

Функции создания и преобразования массивов включают:

- ARRAY MERGE (TSOURCE, FSOURCE, MASK) — производит слияние двух массивов в один по массиву-маске MASK;
- [VECTOR] PACK (ARRAY, MASK[, VECTOR]) — "упаковка" элементов ARRAY, соответствующих истинным значениям массива-маски MASK в вектор VECTOR;
- ARRAY UNPACK (VECTOR, MASK, FIELD) - "распаковка" массива ARRAY из вектора VECTOR по массиву-маске MASK. Недостающие элементы заполняются элементами FIELD, если это массив, или числом FIELD, если это скаляр;
- ARRAY RESHAPE (SOURCE, SHAPE [, PAD] [, ORDER]) — Переформирование массива SOURCE по форме, определяемой одномерным массивом целых чисел SHAPE. PAD представляет собой массив — дополнение массива SOURCE до требуемого размера, а ORDER определяет порядок следования индексов массива-результата, если порядок не должен совпадать с порядком индексов массива SHAPE;

- ARRAY SPREAD (SOURCE, DIM, NCOPIES) — копирует исходный массив NCOPIES раз, возвращая массив на единицу большего ранга и размещая полученные копии в результирующем массиве по измерению DIM;

- ARRAY CSHIFT (ARRAY, SHIFT [, DIM]) — невытисняющий сдвиг массива с шагом SHIFT по индексу DIM. Если DIM не задано, принимается DIM = 1. Если SHIFT — массив целых чисел, размер которого равен рангу массива ARRAY - 1, он рассматривается как массив шагов сдвига по разным измерениям;
- ARRAY EOSHI FT (ARRAY, SHIFT [, BOUNDARY] [, DIM]) — вытесняющий сдвиг с заполнением образующихся пропусков граничными значениями, если они заданы, и нулями — в противном случае.

Функции редукции массивов принимают массив в качестве входного параметра и возвращают скалярный результат или одномерный массив результатов, размер которого равен числу измерений массива.

Наличие в нижеперечисленных подпрограммах необязательного параметра DIM приводит к тому, что исходный массив разбивается на сечения, проходящие по индексу DIM, и в результате получается массив значений для всех полученных сечений. Необязательный аргумент MASK распространяет действие функции только на те элементы массива, для которых соответствующий элемент массива MASK имеет значение .TRUE.

Вот перечень функций редукции:

- LOGICAL ALL (MASK [, DIM]) - возвращает логическое значение "истина", если все элементы логического массива MASK истинны или его размер равен нулю;
- LOGICAL ANY (MASK [, DIM]) — "истина", если хотя бы один элемент массива MASK имеет значение .TRUE .;
- INTEGER COUNT (MASK [, DIM]) — возвращает стандартную целую величину, равную числу элементов истинных элементов массива MASK;
- MAXVAL(ARRAY[, DIM][, MASK]) — возвращает значение максимального элемента целого или вещественного массива и максимальное по абсолютной величине отрицательное число для массива нулевого размера;
- MINVAL(ARRAY[, DIM] [,' MASK]) — возвращает значение минимального элемента целого или вещественного массива и максимальное положительное число для массива нулевого размера;
- PRODUCT (ARRAY [, DIM][, MASK]) - возвращает произведение элементов целого, вещественного или комплексного массива или 1, если массив ARRAY имеет нулевой размер;
- SUM (ARRAY [, DIM] [,- MASK]) — возвращает сумму элементов целого, вещественного или комплексного массива или 0, если массив ARRAY имеет нулевой размер;
- INTEGER MAXLOC(ARRAY!, MASK]) — возвращает в виде массива целых чисел набор индексов наибольшего элемента в массиве;
- INTEGER MINLOC (ARRAY [, MASK]) — возвращает в виде массива целых чисел

набор индексов наименьшего элемента в массиве. В функциях MAXLOC и MINLOC значения индексов рассчитываются так, как если бы нижние границы по всем измерениям равнялись 1. Если экстремальных значений несколько, выбирается первый по порядку следования элементов.

Функции, производящие операции с векторами и матрицами, включают операции векторного, матрично-векторного и матричного умножения. Сложение матриц (совместимых) выполняется оператором +. Имеются также подпрограммы:

- DOT_PRODUCT(VECTOR_A, VECTOR_B) — Возвращает SUM(VECTOR_A * VECTOR_B),

если VECTOR_A целого или вещественного типа, SUM(CONJG(VECTOR_A) *

VECTOR_B), если VECTOR_A комплексного типа и ANY(VECTOR_A.AND.

VECTOR_B) , если VECTOR_B логического типа;

- MATRIX MATMUL (MATRIX_ A,MATRIX_B) -- возвращает результат перемножения исходных матриц, зависящий от их формы. Возможны три случая -матрица-матрица, вектор-матрица и матрица-вектор, функция сама выбирает алгоритм, соответствующий исходным данным.

Директивы HPF

Мы уже отмечали, что в основе модели параллелизма, реализованной в HPF, лежит параллелизм данных. В предположении векторных и массивно-параллельных компьютеров разрабатывались встроенные функции базового языка FORTRAN-90. Основа директив HPF — распределение данных. Средства управления процессами в HPF отсутствуют, и параллелизм задач отсутствует вместе с ними.

Инструкции и встроенные функции HPF

Синтаксис инструкций HPF делает их незаметными для базового языка. Они выглядят как комментарии FORTRAN-90:

!HPF$ <директива_HPF>

HPF практически не вмешивается в действия программы, он только распределяет данные. Программу с директивами HPF можно откомпилировать компилятором для FORTRAN-90, и никаких изменений не потребуется, чтобы компиляция прошла нормально. Можно вводить директивы HPF и в форме комментариев FORTRAN-11:

CHPF$ <директива_HPF>,

где с располагается в первой позиции строки, но первая форма, с нашей точки зрения, предпочтительнее.

Строка-директива может размещаться на двух и более строках, неоконченная строка должна заканчиваться знаком &, а строка-продолжение директивы должна начинаться также с IHPF$.

Концептуальной в HPF является директива PROCESSORS:

!HPF$ PROCESSORS, DIMENSION(:, :) :: P2

!HPF$ PROCESSORS, DIMENSION;:) :: P1 !HPF$ PROCESSORS P(4)

Эта директива вводит описания *псевдомассивов процессоров* — виртуальных конструкций, вообще говоря, не имеющих никакой связи с реальным набором физических процессоров. Одна программа может описывать несколько наборов процессоров. Дело в том, что массивы, а данные, как правило, размещаются в массивах, могут распределяться только по набору процессоров того же ранга, что и распределяемые массивы. Для распределения одномерных массивов необходимо описать одномерный набор процессоров, для распределения матриц понадобится двумерный массив. Таким образом, директива PROCESSORS вводит набор процессоров в виде массивов различной формы. Распределение данных проектируется на стадии компиляции программы тогда, когда массив физических процессоров недоступен. Однако

HPF устроен так, что реальный набор процессоров во время исполнения программы доступен и определен.

Две встроенные функции HPF позволяют получить сведения о наборе процессоров:

- NUMBER_OF_PROCESSORS () — возвращает число физических процессоров;
- PROCESSORS_SHAPE() — определяет форму физического массива процессоров.

Обе встроенные функции доступны уже на уровне директив распределения HPF, но не могут вызываться в операторах описания базовой программы:

!HPF$ PROCESSORS P1(NUMBER_OF_PROCESSORS()) !HPF$ PROCESSORS P2(4,NUMBER_OF_PROCESSORS())

допустимые директивы, а инструкция:

INTEGER, PARAMETER :: NumberOfPrs = NUMBER_OF_PROCESSORS()

недопустима.

Распределение данных

Директива *DISTRIBUTE*

Синтаксис директивы распределения в общем случае таков:

!HPF$ DISTRIBUTE [(способ_распределения)] &

[ONTO псевдомассив_процессоров] :: список_распределяемых_массивов

Пример:

!HPF$ DISTRIBUTE (CYCLIC) ONTO PI :: B

Можно использовать и более близкую к традиционной форме операторов описания запись. Например, следующая директива распределяет блоками двумерный массив A по псевдомассиву процессоров P2:

!HPF$ DISTRIBUTE A(BLOCK, BLOCK) ONTO P2

Если способ распределения в директиве DISTRIBUTE опущен, принимается блочное распределение. Если опущена конструкция ONTO, распределение выполняется по набору процессоров, принятому по умолчанию, в котором число процессоров равно числу физических процессоров, а форма псевдомассива соответствует форме распределяемого массива. Наиболее содержательным аспектом распределения данных является выбор способа распределения.

Неудачно произведенное распределение может привести к катастрофическому росту обмена и фатальному падению производительности. Общим дексами. Блочное распределение сводит к минимуму обмен при программировании операций типа:

a(I) = a(I - 2) * *(a(*I - 1) + a(1) + a(I + 1))

т. к. большая часть операций будет проделываться с элементами, принадлежащими одному блоку.

Определение параметров размера блока или шага цикла в задании способа распределения почти нивелирует разницу между способами распределения. Действительно, и BLOCK (m), и CYCLIC (m) приведут к тому, что элементы будут распределяться блоками по m элементов в каждом, с той лишь разницей, что при блочном распределении все элементы должны быть распределены по процессорам за один круг, иначе компилятор выдаст сообщение об ошибке. При циклическом же распределении ошибки не будет и раздача элементов повториться столько раз, сколько нужно для раздачи всех элементов. CYCLIC (m) представляется наиболее удачной версией распределения: оно обеспечивает и отношение соседства, и

равномерность загрузки. Однако для сохранения информации о распределении элементов при этом способе требуется в несколько раз больше триплетов индексов, чем при других видах распределения. Это может свести на нет все преимущества от использования CYCLIC(m).

Распределение BLOCK (m) полезно в том случае, когда нужно занять данными только часть процессоров, оставив остальные процессоры свободными. Директива:

!HPF$ DISTRIBUTE (BLOCK(31)) :: A, B

разместит первые 93 элемента массива A на первых трех процессорах PI, 7 оставшихся элементов на 4-м процессоре, остальные 6 процессоров не получат ни одного элемента. Что касается массива в, то все его элементы попадут на первый процессор псевдомассива PI.

Распределение многомерных массивов

Как было сказано выше, размерность массива процессоров, по которому распределяются данные, должна совпадать с размерностью распределяемого массива, а точнее, с числом измерений, в которых данные распределяются. Способы распределения в разных измерениях могут быть различными. Двумерный массив вещественных чисел R можно распределить по двумерному массиву процессоров блочно в обоих направлениях:

REAL, DIMENSIONS, :) :: R(100, 100)

!HPF$ PROCESSORS, DIMENSION(:, :) :: P(2, 2)

!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO P :: R

Процессор с номером (1,1) получит блок элементов массива R с номерами (1:50, 1:50), процессор с номером (2, 1) номера (51:100, 1:50) и т. д. Можно распределить элементы в одном измерении блочно, в другом циклически:

!HPF$ DISTRIBUTE (BLOCK, CYCLIC) ONTO P :: R

Результат не так очевиден, как в предыдущем случае, но поддается и представлению, и описанию:

1) -> P(1, 1), R(l, 2) -> P(1, 2), R(l, 3) -> P(1, 1), , 4) -> P(1, 2) ...

R(51, 1) -> P(2,l),..., R(51, 100) -> P(2, 2)

Можно отказаться от распределения элементов в одном измерении, определяя принадлежность элемента по другому индексу. Отказ от распределения в указанном

измерении называется *коллапсом измерения,* обозначается звездочкой (*) на месте спецификатора способа распределения, и требует псевдомассив процессоров рангом на единицу меньше:

!HPF$ PROCESSORS, DIMENSION^) ::

PI (4) !HPF$ DISTRIBUTE (BLOCK, *) ONTO PI : : R

При таком распределении все строки массива R с номерами 1—25 попадут на процессор P1 (1) , на втором окажутся строки с номерами 26—50 и т. д.

Распределение динамических массивов

Размер массива данных не всегда известен на стадии компиляции. В этом случае память под такой массив выделяется динамически, инструкцией ALLOCATE. Динамические массивы могут быть распределены, но привязка их элементов к процессорам выполняется во время исполнения кода, после операции ALLOCATE:

REAL, ALLOCATABLE, DIMENSION (: , :) : :

A INTEGER : : ierr !HPF$ DISTRIBUTE A(*, BLOCK)

ALLOCATE (A (20, 100), STAT = ierr)

Здесь A автоматически распределяется в соответствии с заданным распределением. После операции DEALLOCATE (A) распределение аннулируется.

Скалярные переменные

Скалярные переменные копируются (реплицируются) для каждого процессора и компилятор отвечает за когерентность (согласованность) их значений. Поэтому, если скаляру в ходе выполнения программы будет присвоена переменная, привязанная к одному из процессоров, это вызовет посылку

широковещательных сообщений с целью синхронизации обновленного значения на всех процессорах:

REAL,DIMENSION(100, 100) :: X REAL ::

Scal !HPF$ DISTRIBUTE (BLOCK, BLOCK) :: X

Seal = X(i, j)

Если без таких присваиваний невозможно обойтись, рекомендуется завести псевдомассив с этим скаляром и распределить его циклически по всем процессорам.

Выравнивание. Директива *ALIGN*

Распределение элементов массивов по процессорам может быть не столь важным, как распределение элементов одних массивов по отношению к другим массивам. Важно, чтобы все компоненты одного выражения были привязаны к одному и тому же процессору. Привязка по образцу уже описанного распределения называется *выравниванием* и вводится директивой ALIGN. Ее синтаксис:

!HPF$ ALIGN(форма_выравниваемого_массива) &

WITH образец_выравнивания :: список_выравниваемых_массивов

Есть две формы записи директивы выравнивания: общая и элементная. В общей форме директива записывается так:

!HPF$ ALIGN (:, :) WITH T(:, :) :: A, B, C

В результате выполнения директивы элементы массивов A, в, с с индексами (i, j) будут привязаны к тому же процессору, что и элемент т(1, j). Директиву DISTRIBUTE можно применить только к массиву т, остальные массивы распределятся так же, как т. Общая форма записи директивы ALIGN требует согласованности формы выравниваемых массивов, хотя выравнивание может производиться и тогда, когда это условие не выполняется. Элементная форма директивы ALIGN обходит это ограничение:

REAL,DIMENSION(10) :: A, B, C !

HPF$ ALIGN (j) WITH C(j) :: A, B

Элементы массивов A и в с индексом j должны быть расположены там же, где находится элемент массива сие тем же индексом. Требуется только, чтобы в массивах A и в не оказалось больше элементов, чем в массиве с.

Примеры выравнивания. Выравнивание может быть очень разнообразным и даже весьма изощренным. Приведем несколько примеров.

Транспонированное выравнивание:

REAL, DIMENSION(10, 10) ::

A, B !HPF$ ALIGN A(i, :) WITH B(:, i)

Выравнивание с шагом:

REAL, DIMENSION(5) ::

D REAL, DIMENSION(10) ::

E !HPF$ ALIGN D(:)'WITH E(I::2)

или:

!HPF$ ALIGN D(I) WITH E(I * 2-1)

Первый элемент массива о будет совмещен с первым элементом массива Е, второй с третьим, третий с пятым и т. д.

Обратное пошаговое выравнивание:

!HPF$ ALIGN D(:) WITH E(UBOUND(E)::-2)

или:

!HPF$ ALIGN D(I) WITH E(2 + UBOUND(E) - I * 2)

Выравнивание динамических массивов. При выравнивании динамических массивов необходимо помнить, что выравнивание выполняется при операции выделения памяти для массива и нельзя выравнивать регулярный массив по динамическому.

Коллапс и репликация измерений при выравнивании. Рассмотрим следующий пример выравнивания:

REAL, DIMENSION(10) :: Y REAL, DIMENSION(10, 10) ::

X !HPF$ ALIGN(*, :) WITH Y(:) :: X

Коллапс измерения при выравнивании следует понимать так: все элементы i-го столбца матрицы x должны быть привязаны к тому процессору, который обладает i-м элементом вектора y.

Теперь для тех же массивов x и Y рассмотрим следующую директиву:

!HPF$ ALIGN Y(:) WITH X(*, :)

Это реплика-копирование одного и того же элемента несколько раз для разных процессоров. В данном случае i-и элемент вектора Y будет скопирован на все процессоры, на которых расположены элементы i-го столбца матрицы x. Сознательная репликация — мощный инструмент в руках программиста, средство предотвращения многих неприятных ситуаций с неконтролируемой передачей данных.

Шаблоны выравнивания.

Директива *TEMPLATE.* Кроме псевдомассива процессоров, HPF определяет еще один специфический объект — *псевдомассив-образец.* Образец напоминает массив, но не является таковым. Он имеет размер, ранг, форму, но не имеет элементов и, соответственно, не занимает места в памяти.

Образец не является элементом FORTRAN-90, но его имя не должно совпадать с именем какого бы то ни было объекта, определенного в языке FORTRAN. Образец должен быть продекларирован и статически определен. К образцу могут быть применены директивы распределения и выравнивания. Ради использования директивы ALIGN в блоке WITH образцы и были задуманы. Пример:

REAL, DIMENSION(10) ::

A, B !HPF$ TEMPLATE, DIMENSION(10) ::

T !HPF$ DISTRIBUTE (BLOCK). ::

T !HPF$ ALIGN(:) WITH T(:) :: A, B

Трудно придумать пример распределения данных, в котором невозможно было бы обойтись без образцов. Однако использование образцов позволяет разделить директивы HPF и описания FORTRAN, что делает HPF более удобным. Не распределенный явным образом массив подвергается распределению по умолчанию. Как правило (но не всегда!), это реплика. Тогда, когда нужно провести принудительное копирование, используются образцы. Например:

REAL, DIMENSION(100, 100) :: A

!HPF$ PROCESSORS, DIMENSION(NUMBER_OF_PROCESSORS()) ::

P !HPF$ TEMPLATE, DIMENSION(NUMBER_OF_PROCESSORS()) ::

T !HPF$ ALIGN A(*, *) WITH T(*) !HPF$ DISTRIBUTE (BLOCK) :: T

Процедуры и распределение данных

Информация о распределении данных не может быть передана в подпрограмму — для этого в базовом языке нет средств. Директивы распределения данных могут размещаться в любой подпрограмме и будут выполнены компилятором. При передаче управления подпрограмме данные могут быть распределены заново, надо ли говорить, к какому "чудесному" эффекту это может привести?

Информацию о распределении глобальных данных можно передать в подпрограмму, поместив директивы распределения в модуль и включив этот модуль во все компоненты программы, оперирующие с глобальными переменными. А как передавать

формальные параметры и информацию об их распределении?

В HPF существует три различных подхода к распределению формальных параметров:

- *предписательный* — распределение должно быть выполнено;
- *описательный* — распределение уже выполнено;
- *воспроизводящий* — распределение формальных аргументов должно быть воспроизведено по распределению фактического.

Наиболее естественным кажется последний, воспроизводящий (transcriptive) или наследуемый подход к распределению формальных аргументов. Однако в процессе развития компиляторов HPF стало очевидно, что реализовать воспроизводящий метод распределения формальных аргументов эффективно невозможно. В языке FORTRAN-90 вместе с массивами используются их сечения. Компилятор должен предусматривать слишком много вариантов при передаче многомерных массивов в процедуры в качестве формальных параметров, поэтому рассмотрим два других подхода.

Предписательное распределение выглядит точно так же, как и распределение в основной программе; директивы предписательного распределения будут выполняться при передаче управления в подпрограмму. По выходе из подпрограммы фактические аргументы будут снова распределены так, как они были распределены до входа в подпрограмму. Уменьшить количество перераспределений можно, используя атрибут INTENT формальных параметров, двойному перераспределению будут подвергнуты только параметры, имеющие атрибут INTENT (INOUT). Вынесение информации о распределении данных в процедуре в явный интерфейс тоже косвенно способствует оптимизации перераспределения при вызове подпрограмм.

Пример предписательного распределения:

SUBROUTINE subprog(A, B, RES)

IMPLICIT NONE

REAL, DIMENSIONS, :), INTENT (IN) : : A, B

REAL, DIMENSIONS, :), INTENT (OUT) :: RES

!HPF$ PROCESSORS, DIMENSION(2, 2) ::

P !HPF$ TEMPLATE, DIMENSIONS, 6) : :

T !HPF$ ALIGN (:, :) WITH T(:, :) ::

A, B, RES !HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO P :: T

END SUBROUTINE subprog

Смысл описательного распределения иной. При описательном распределении передается информация об уже существующей привязке данных. Перераспределение производится только в том случае, когда фактическое распределение не совпадает с описанным. Процедура subprog из предыдущего примера в описательном варианте будет выглядеть так:

SUBROUTINE subprog(A, B, RES)

IMPLICIT NONE

REAL, DIMENSIONS, :), INTENT (IN) : : A, B

REAL, DIMENSION):, :), INTENT(OUT) :: RES

!HPF$ PROCESSORS, DIMENSION(2, 2) ::

P !HPF$ TEMPLATE, DIMENSION(4, 6) ::

T !HPF$ ALIGN (:, :) WITH *T (:, :) :: A, B, RES

!HPF$ DISTRIBUTE *(BLOCK,BLOCK) ONTO *P : : T

END SUBROUTINE subprog

Знаком * помечаются части директив, уже имеющие место. То есть A, в и RES уже выровнены по т, а т уже распределено (BLOCK,- BLOCK) по набору процессоров P. Если * перед р опустить, смысл директивы изменится так: т распределен (BLOCK, BLOCK) , но не по р.

Описательный стиль директив распределения является предпочтительным. Настоятельно рекомендуется вносить описание директив распределения в интерфейсы. Используя модули для передачи директив распределения, следует помнить, что объекты TEMPLATE и PROCESSORS не являются объектами FORTRAN, как инструкции USE, вследствие чего они не могут быть переименованы посредством конструкции USE ONLY.

Параллелизм исполнения в HPF

Все ранее рассматривавшиеся директивы HPF касались только распределения (привязки) данных к процессорам, не выражая параллелизма действий. Параллелизм действий в HPF проявляется, главным образом, в исполнении встроенных функций FORTRAN-90, таких как назначения массивов. Там, где не включается встроенный параллелизм базового языка, применяются директивы распараллеливания исполнения

HPF. Они относятся к циклам и к назначениям массивов, использующих обращения к индексам. Это инструкции INDEPENDENT, NEW, PURE И FORALL.

Инструкция *FORALL*

Инструкция настолько "прижилась", что была включена в следующую версию FORTRAN — FORTRAN-95. В реализацию PortlandGroup FORTRAN-90 эта инструкция тоже включена, для ее применения не требуется приставка !HPF$. Мощных средств работы с массивами оказалось недостаточно, встроенные процедуры, принимающие массивы в качестве аргументов, не могут производить действий с индексами массивов. Использование циклов оставлено для исключительно последовательных действий; можно представить, как оно замедляет исполнение на компьютере векторного типа, когда итерации цикла проводятся строго упорядоченно.

Синтаксис инструкции:

FORALL (список_назначений_индексов

[, скалярная_маска]) & оператор_назначения

Например:

FORALL (I = l:n, j = l:m, A(i, j).NE.0) A(i, j) = 1 / A(i, j)

Если назначений несколько, они заключаются в скобки FORALL. . .END FORALL:

FORALL (I = l:n:2, j = n:l - 2, A(i, j).NE.0) A(i, j) = 1 / A(i, j)

}

END FORALL

Выполнение инструкции FORALL происходит в четыре этапа:

1. Вычисляется индексное выражение.

2. Вычисляется маска для всех индексов.

3. Для всех элементов с маской .TRUE, вычисляются правые части выражения присваивания.

4. Производится назначение.

После каждого этапа проводится синхронизация.

Смысл инструкции FORALL отличается от смысла тех же действий, выполняемых

внутри цикла. Сравните цикл:

DO I = 2, п - 1

END DO

И

FORALL (I = 2.:n - 1) a(i) = a(I - 1) + a(i) + a(I + 1)

PURE-процедуры

Внутри конструкции FORALL могут вызываться процедуры, определенные программистом как PURE (чистые, правильные). PURE-процедуры не должны производить сторонних эффектов, для этого они должны удовлетворять следующим условиям:

- не производить ввода/вывода и операций с памятью;
- не менять глобальное состояние программы;
- не содержать операторов PAUSE и STOP;
- не иметь локальных переменных с атрибутом SAVE;
- не выравнивать локальные переменные по глобальным данным;
- содержать обращения только к PURE-процедурам.

Все встроенные функции языка FORTRAN-90 являются PURE-процедурами.

Программист сам решает, является ли написанная им процедура PURE-процедурой, компилятор только проверяет формальную сторону допустимости того или иного вызова.

Директива *INDEPENDENT*

Директива INDEPENDENT применяется к операторам цикла и FORALL. В тексте программы инструкция помещается непосредственно перед тем оператором, к которому она относится, например:

!HPF$ INDEPENDENT DO I = 1, n

a(i) = b(i) + c(i) * g(i) ENDDO

или:

!HPF$ INDEPENDENT

FORALL (I = l:n, j = l:m, A(I, j).GT.O) A(I, .j) = REAL(I + j)

Для операторов цикла директива INDEPENDENT означает независимость итераций друг от друга. Итерации могут выполняться в произвольном порядке, тогда компилятор может поручить их исполнение различным процессорам. Ответственность за принятие решения возлагается на программиста, автоматически циклы не распараллеливаются. Чтобы понять, можно ли цикл отметить как INDEPENDENT, нужно поменять местами верхний и нижний пределы цикла, а знак шага изменить на противоположный. Если от такой перестановки результат вычислений не изменится, можно смело ставить перед циклом директиву INDEPENDENT.

В цикле INDEPENDENT допускаются вызовы только PURE-процедур. Директива INDEPENDENT для FORALL означает снятие синхронизации между третьим и четвертым этапами выполнения FORALL.

Иногда добиться независимости итераций цикла помогает создание дополнительных экземпляров скалярных переменных для каждой итерации. Создание переменных инициируется конструкцией NEW в директиве INDEPENDENT:

!HPF$ INDEPENDENT, NEWJrl, r2) DO j = 1, n

rl = SIN(a(j)) r2 = COS(a(j)) a(j) = rl * rl - r2 * r2 END DO

Для каждого j компилятор создаст собственный экземпляр переменных. Переменные, отмеченные NEW, не могут:

- использоваться вне цикла без переопределения;
- быть необязательным аргументом или указателем;
- иметь атрибут SAVE.

Недопустимо использование переменных NEW в FORALL, т. к. там они не имеют смысла. Конструкция NEW неприменима к FORALL.

Вопросы и задания для самостоятельной работы

Упражнения 1—6 посвящены программированию на языке FORTRAN-90, остальные — на HPF.

1. На примере программы FACT (листинг 8.1), вычисляющей значение факториала заданного числа, рассмотрите структуру главной программы с внутренней рекурсивной функцией.

Листинг 8.1. Программа FACT

PROGRAM FACT

IMPLICIT NONE

```
INTEGER N, FAC

PRINT *, 'Введите число N:'

READ(*, *) N

IF(N>0) THEN

FAC=FACTORIAL(N) ELSE

STOP 'Недопустимое значение аргумента' ENDIF

PRINT *,'N! = ', FAC CONTAINS

INTEGER RECURSIVE FUNCTION

FACTORIAL(I) RESULT(FRES)

INTEGER I, IR

IF (I=1) THEN FRES = 1

ELSE FRES = I * FACTORIAL(I - 1)

ENDIF

END FUNCTION FACTORIAL END

PROGRAM FACT
```

Что нужно изменить в программе, чтобы внутренняя подпрограмма стала внешней? Определите, каков диапазон исходных данных для программы FACT? Что можно сделать для расширения диапазона исходных значений? Напишите аналогичную программу, вычисляющую число e с заданной точностью.

2. На примере программы CIRCLES (листинг 8.2) рассмотрите описание производного типа COLORED_CIRCLE (цветной кружок), помещенное в модуль COLORED. Обратите внимание на определение операции + (соединение) для производного типа и использование конструктора структур.

Листинг 8.2. Программа CIRCLES

```
MODULE COLORED TYPE COLORED_CIRCLE

REAL CX, CY, RADIUS
```

```
CHARACTER(LEN = 5) COLOR END TYPE NTERFACE OPERATOR(+)
MODULE PROCEDURE UNITE END INTERFACE CONTAINS
FUNCTION UNITE(A, B)
TYPE(COLORED_CIRCLE) UNITE
TYPE(COLORED_CIRCLE), INTENT(IN) :: A, B
IF(A%RADIUS >= B%RADIUS) THEN UNITE%COLOR = A%COLOR
ELSE UNITE%COLOR = B%COLOR
ENDIF
UNITE%CX = A%CX 4- B%CX
UNITE%CY = A%CY + B%CY
UNITE%RADIUS = A%RADIUS + B%RADIUS
END FUNCTION UNITE END MODULE COLORED
FUNCTION UNITE(A, B) TYPE(COLORED_CIRCLE)
UNITE TYPE(COLORED_CIRCLE), INTENT(IN) ::
A, B IF(A%RADIUS >= B%RADIUS) THEN
UNITE%COLOR = A%COLOR ELSE
UNITE%COLOR = B%COLOR ENDIF
UNITE%CX = A%CX + B%CX UNITE%CY = A%CY + B%CY
UNITE%RADIUS = A%RADIUS + B%RADIUS END
FUNCTION UNITE END MODULE COLORED
PROGRAM CIRCLES
USE COLORED
IMPLICIT NONE
TYPE(COLORED_CIRCLE) :: big_red, little_green, result
```

big_red = COLORED_CIRCLE(0, 0, 10, 'red')

little_green = COLORED_CIRCLE (2, *2,* 5, 'green').

result = big_red + little_green

PRINT *,'COLORED_CIRCLE result:', ' COLOR ', result%COLOR

PRINT *,'CX=', result%CX,'

CY=', result%CY, '

RADIUS=', result%RADIUS

END PROGRAM circles

Можно ли поместить описание производного типа и операции сложения переменных этого типа непосредственно в программу CIRCLES?

3. Создайте, не пользуясь встроенным типом COMPLEX, описание производного типа, поведение которого совпадает с поведением комплексных чисел. Напишите небольшую программу, использующую переменные описанного типа.

4. Не используя обращение к встроенной функции перемножения матриц MATMUL, напишите свою собственную версию этой функции. Каким образом можно добиться соединения под одним именем обращения к трем различным подпрограммам?

5. Программа SHIFTS (листинг 8.3) заполняет одномерный массив натуральными числами, "размножает" его заданное число раз и выполняет операцию сдвига по каждому из измерений полученного двумерного массива.

Листинг 8.3. Программа SHIFTS

PROGRAM SHIFTS IMPLICIT NONE

INTEGER, ALLOCATABLE, DIMENSION(:) ::

S INTEGER :: N, NR, I, J, IERR INTEGER, ALLOCATABLE, DIMENSIONS, :) ::

R PRINT *, 'Введите размер исходного массива:' READ(*, *) N

PRINT *, 'Введите число копий:' READ(*, *) NR

ALLOCATE(S(N), STAT = IERR) IF(IERR == 0)

THEN ALLOCATE(R(N, NR),

```
STAT = IERR) IF(IERR == 0) THEN

DO I = 1, N S(I)=I;

ENDDO

PRINT *, 'Исходный массив:'

PRINT *, 'S= ', S(1:N)

R = SPREAD(S, 2, NR)

PRINT *, 'Скопированный массив:'

DO I = 1, NR

PRINT *, 'R(', I, ', :) ', R(I, : ) ENDDO

PRINT *, 'Преобразованный массив:'

DO I = 1, NR R(:, I) = R(:, I) * I PRINT *, 'R(', I, ',:) ', R(I, :)

ENDDO

R = CSHIFT(R, 1) PRINT *, 'Сдвиг в измерении 1;'

DO I = 1, NR

PRINT *, 'R(', I, ', :) ', R(I, :) ENDDO

R = CSHIFT{R, 1, 2) PRINT *, 'Сдвиг в измерении 2:'

DO I = 1, NR

PRINT *, 'R(', I, ', :) ', R(I, :) ENDDO ELSE

PRINT *, 'ALLOCATE R, IERR=', IERR ENDIF ELSE

PRINT *, 'ALLOCATE R, IERR=', IERR ENDIF

END PROGRAM SHIFTS
```

Преобразуйте программу так, чтобы в результате получался трехмерный массив, а сдвиг производился бы на заданное число шагов для каждого измерения.

6. Каким образом, не прибегая к явному перебору, в многомерном массиве определить значения элементов, окружающих элемент с заданным индексом?

7. Опишите, как будет распределен массив A:

REAL A(100)

!HPF$ PROCESSORS P(10) !HPF$ DISTRIBUTE A(BLOCK) ONTO P

Как изменится распределение, если будет опущена конструкция ONTO?

8. Опишите, как будет распределен массив A:

REAL A(91)

!HPF$ PROCESSORS P(ll) !HPF$ DISTRIBUTE A(CYCLIC) ONTO P

Что произойдет, если в директиве DISTRIBUTE будет опущено указание (CYCLIC)?

9. Опишите, как будет распределен массив A:

REAL A(100, 100) !HPF$ PROCESSORS P(16, 16) !HPF$ DISTRIBUTE A(BLOCK, CYCLIC) ONTO P

Каким будет распределение, если указать параметры блочного и циклического распределений: BLOCK (4) и CYCLIC (4) ?

10. Опишите, как будет распределен массив A:

REAL A(100,100) !HPF$ PROCESSORS P(1б) !HPF$ DISTRIBUTE A(BLOCK, *) ONTO P

Объясните, почему в последнем примере двумерный массив распределяется по одномерному набору процессоров?

11. Напишите программу вычисления сумм всех элементов в строке двумерного массива. Выберите и опишите подходящее для поставленной задачи распределение элементов массива по набору из 4-х процессоров.

12. Напишите программу диагонализации матрицы методом Гаусса, оформив процесс диагонализации в виде подпрограммы. Выполните распределение данных в главной программе, затем передайте информацию о распределении формальных параметров в подпрограмму.

13. Используя директиву ALIGN, измените программу из упражнения 12 так, чтобы распределение вектора результатов было связано с распределением исходного массива. Можно ли распределить элементы исходного массива циклически?

14. Используя директиву ALIGN, напишите программу, вычисляющую произведение матрицы на вектор. Примените для распределения выравнивание и шаблон.

15. Чем, на ваш взгляд, различаются циклы:

```
DO I = 2, N - 1
A(I) - A(I-1) + A(1) + A(1 + 1)
END DO
```

И

```
FORALL (I = 2:N-1) A (I) = A(I - 1) + A(I + 1)
```

Имеет ли смысл инструкция FORALL без директивы INDEPENDENT?

ПРИЛОЖЕНИЕ 1.

Средства отладки и мониторинга параллельных MPI-программ

MPE — многопроцессорное окружение

Многопроцессорное окружение (Multi-Processing Environment, MPE) предоставляет программисту полный набор средств анализа производительности MPI-программ. Программы MPE создают протоколы исполнения (log-файлы) MPI-программ для постпроцессорной обработки с помощью графических инструментов, поставляемых вместе с MPE.

Основными компонентами MPE являются:

- набор профилирующих библиотек;
- библиотека параллельной анимации для X Windows;
- программы "депараллелизации" сегментов параллельного кода;
- программы установки отладчиков.

MPE входит в состав дистрибутива MPICH и, если конфигурирование MPICH не было выполнено с флагом —without-mpe, основные компоненты MPE инсталлируются вместе с ним.

Создание log-файлов

MPE предоставляет несколько способов генерации log-файлов, описывающих ход исполнения вычислений. Log-файлы можно создавать как вручную, при включении вызовов программ MPE в MPI-программы, так и автоматически, при связывании с

необходимыми MPE библиотеками, или же комбинацией обоих методов. Кроме того, программист имеет возможность создавать адаптированные для данного приложения log-файлы. Для генерации протоколов достаточно включить в параметры компилятора, создающего исполняемый файл программы, ключ с именем соответствующей библиотеки.

Форматы log-файлов

В настоящее время MPE поддерживает три различных формата log-файлов: ALOG, CLOG и SLOG. ALOG сохраняется только для обратной совместимости и больше не разрабатывается. CLOG представляет собой простой набор событий с одним штампом времени. Программа Jumpshot-2 предназначена для разбора файлов в формате CLOG. SLOG-формат (Scalable LOG) основан на дважды хронометрированных состояниях. Расширяемость SLOG-формата обеспечивается разделением всех состояний log-файла на фреймы данных, достаточно малых для эффективной обработки программой визуализации Jumpshot-3. В то же время, при сравнении двух соседних фреймов, состояния и стрелки выглядят неразрывными. Jumpshot-3 способен обрабатывать файлы формата SLOG, измеряемые гигабайтами.

Параллельная графика

MPE включает набор программ, которые строят простые изображения в системе X Windows. Пример использования MPE графики для MPICH находится В каталоге mpich/mpe/contrib/mandel.

Используя файл Makefile из этого каталога, можно получить исполняемый файл pmandei. Команда mpirun -np 4 pmandei при исполнении выведет на экран изображение классического множества Мандельброта.

Другие MPE-программы

Иногда во время исполнения параллельной программы возникает необходимость убедиться в том, что один из процессоров делает именно то, что нужно. Программы MPE_Seq_begin И MPE_Seq_end позволяет создавать "последовательные участки" в параллельной программе.

Стандарт MPI разрешает пользователю легко определять программу обработки ошибок, обнаруженных MPI. Часто в таком случае хочется заставить программу запустить отладчик, чтобы немедленно диагностировать проблему. Если MPE конфигурировалось с опцией -mpedbg, обработчик ошибок из MPE_Errors_call_dbx_in_xterm позволит выполнить именно это. Кроме того, при данной опции конфигурации можно скомпилировать библиотеки MPE с включением кода отладки.

Библиотеки профилирования

Интерфейс профилирования в MPI обеспечивает возможность легко добавлять инструменты анализа производительности для любой реализации MPI.

В первую очередь используются программы профилирующих библиотек, создающие файлы протоколов исполнения (log-файлы). MPE предлагает три профилирующие библиотеки.

- *Библиотека хронометрирования MPI-процедур* — очень простая библиотека. Профилирующая версия каждого вызова MPI_XXX подпрограммы вызывает PMPi_wtime (который проставляет штамп времени) до и после каждого вызова соответствующей PMPI_XXX процедуры. Время суммируется в каждом процессе и записывается в отдельном файле на процесс в профилирующей версии MPi_Finaiize. Файлы остаются для последующего включения как в общий, так и в попроцессные отчеты. Вложенные вызовы не обрабатываются.
- *Библиотека автоматического протоколирования,* которая генерирует log-файлы. Во время исполнения программы обращения к MpE_Log_cvent сохраняют записи о событиях в специальных областях памяти, которые объединяются в MPi_Finalize. MPi_Pcontroi можно использовать для приостановки и рестарта операций протоколирования. Вызовы MPE_Log_event делаются автоматически для каждого MP1-вызова.
- *Библиотека параллельной анимации.* Графическая библиотека MPE содержит программы, которые дают возможность нескольким процессам разделять X-дисплей, не связанный ни с каким определенным процессом.

Адаптированное протоколирование

Кроме использования предопределенных библиотек протоколирования, записывающих каждый вызов MPI, протоколирующие вызовы MPE могут быть вставлены в M Pi-программу пользователя там, где он хочет определять и записывать состояния. Состояния могут быть вложенными; позволяется определять состояние, содержащее несколько MPI-вызовов, и отображать как состояние, определенное пользователем, так и MPI-операции, содержащиеся в нем. Вызов MPE_Log_get_event_nuinber нужно использовать для получения уникальных номеров событий, а процедуры MPE_Describe_state и MPE_Log_event — для описания определенных пользователем состояний.

Листинг П1.1. Пример адаптированного протоколирования

```
int eventID_begin, eventID_end;

eventID_begin = MPE_Log_get_event_number ( ) ; eventID_end =
MPE_Log_get_event_number ( ) ;
```

```
MPE_Describe_state ( eventID_begin, eventID_end, "Amult", "bluegreen") ;

MyAmult ( Matrix m. Vector v) {

/* Зафиксируем начало события вместе с размером матрицы */

MPE_Log_event ( eventID_begin, m->n, (char *)0);

... Amult code, including MPI calls ...

MPE__Log_event ( eventID_end, 0, (char *)0);

}
```

Log-файл, сгенерированный этим кодом, при визуализации отметит MPI-вызовы, сделанные из подпрограммы MyAmult, сине-зеленым прямоугольником.

Можно комбинировать автоматическое протоколирование с ручным. Автоматическое протоколирование записывает каждый MPI-вызов, что достигается связыванием с библиотекой протоколирования. При ручном протоколировании пользователь сам вставляет необходимые ему вызовы программ MPE вокруг вызовов MPI. Тогда протоколируются только выбранные обращения к MPI. Но если используется комбинация обоих методов, пользователь не должен вызывать MPE_init_iog и MPE_Finish_iog в своей программе. Связанный с библиотекой протоколирования MPi_init вызовет MPE_init_iog, а MPI_Finalize обратится к MPE_Finish_log.

Утилиты MPE

Набор утилит в MPE содержит конвертеры формата log-файлов (например, clog2slog), программу печати log-файлов (slog_print) и оболочку для просмотра log-файлов — logviewer, который по расширению имени файла выбирает соответствующий графический инструмент для показа log-файла. П Конвертеры формата протоколирования:

- **clog2slog** — преобразует log-файл из формата CLOG в формат SLOG. Автоматическая генерация файлов в формате SLOG при задании значения SLOG переменной окружения MPE_LOG__FORMAT может не сработать из-за некорректного поведения некоторых MPI-программ. Конвертер формата в таких случаях дает возможность произвести экстрадиагностику состояния log-файла. Конвертер также разрешает устанавливать некоторые параметры log-файлов, "такие как размер фрейма — части log-файла, отображаемой в окне "Time Line" программы Jumpshot-3. Для некорректных MPI-программ можно увеличить размер фрейма от 64 Кбайт, назначаемых по умолчанию, до больших значений. Для получения полной информации используйте: clog2slog

-h
    - **clog2alog** — конвертер log-файлов из формата CLOG в формат ALOG, поддерживаемый только ради обратной совместимости.
- Программы печати форматированных протоколов:
    - **slog_print** — программа печати на стандартный вывод файлов в формате SLOG. Она служит для проверки содержимого log-файла. Если log-файл слишком велик, slog_print может оказаться бесполезной. Если log-файл не полон, то slog_print также не будет работать. Таким образом, эта программа служит простым тестом для проверки полноты генерации log-файла;
    - **clog_print** — программа печати на стандартный вывод файлов в формате CLOG.
- Селектор программ просмотра:
    - **logviewer** — скрипт, который находит соответствующую расширению имени файла программу для просмотра его содержимого. Например, для log-файла foo.slog, logviewer вызовет Jumpshot-3, чтобы его просмотреть. Jumpshot-3 располагается в каталоге share/.

Переменные окружения

Существуют две переменные окружающей среды, TMPDIR и MPE_LOG_FORMAT, которые нужно установить до начала генерации log-файлов: П MPE_LOG_FORMAT — определяет формат log-файла, генерируемого при исполнении приложения, связанного с библиотеками протоколирования MPE. Разрешены значения CLOG, SLOG и ALOG. Когда значение переменной не установлено, подразумевается CLOG;

- TMPDIR — указывает каталог, используемый как временное хранилище для файлов от каждого процесса. По умолчанию используется каталог /tmp. Если нужно создать очень большой log-файл для продолжительного MPI-задания, следует убедиться, что каталог, заданный TMPDIR, достаточно вместителен для временных log-файлов, уничтожаемых в случае успешного создания объединенного log-файла. Чтобы минимизировать риск переполнения файловых систем протоколами MPI-программ, пользователям настоятельно рекомендуется выбирать в качестве значения TMPDIR локальные файловые системы.

Важно убедиться, что все процессы MPI получают правильное значение переменных окружения. MPICH для самых широко используемых устройств, ch_p4 и ch_shmem, обеспечивает автоматическую передачу переменных окружения порожденным процессам, но для других устройств это выполняется далеко не всегда. В других реализациях MPI пользователи сами должны следить за правильностью передачи переменных окружения.

Примечание

Окончательный объединенный log-файл будет записан в той файловой системе, в которой выполнялся процесс ранга 0.

Присоединение библиотек

Для присоединения программ из библиотек MPE, необходимо в параметрах компилятора или редактора связей указать имя библиотеки, к программам которой производятся обращения. Естественно, что, кроме библиотек MPE, обязательно указывается библиотека MPI: -impich или -impi (в зависимости от имплементации MPI).

Флаги компилятора, присоединяющие библиотеки:

- профилирования: -lmpe -impich;
- протоколирования: -llmpe -lmpe -lmpich;
- Трассировки: -ltmpe -lmpe -lmpich;
- анимации: -lampe -lmpe -lmpich -1X11;
- FORTRAN-TO-C_MPI-оболочка: -lmpe_f2cmpi.

Флаг -impich при использовании отличных от MPICH реализаций MPI должен быть заменен на соответствующий названию библиотеки MPI; вместо -1X11 может использоваться другая графическая библиотека X Windows, а библиотека-оболочка FORTRAN-to-C_MPI в MPICH вызывается как -lfmpich.

В MPICH присоединение библиотек MPE обеспечивается опциями компилирующих скриптов mpicc/mpiCC и mpif77/mpif90: -mpilog, -mpianim, и -mpitrace.

Если связывание производится для MPI-программы, написанной на FORTRAN, флаг -lfmpich должен указываться до флагов профилирующих библиотек:

-lfmpich -lmpe -lmpich

Это разрешает использование программ, применяемых в библиотеках профилирования, и в C, и в FORTRAN-программах. В других реализациях MPI флаг -lmpe_f2cmpi также должен предшествовать флагам профилирующих библиотек, например: -lmpe_f2cmpi -llmpe -lmpe -lpmpi -lmpi

Автоматическая генерация библиотек

Все библиотеки MPE построены по одной, весьма несложной, схеме. Все вызовы MPI-процедур переписываются так, чтобы представлять собой обернутый несколькими операторами вызов соответствующей PMPI-npo-цедуры. Сначала нужно написать профилированные версии MPi_init и MPi_Finalize. Другие MPI-процедуры записываются аналогичным образом, приблизительно так, как в листинге П1.2.

Листинг П1. 2. Схема профилированной MPI-процедуры

int MPI_XXX (...)

{

какие-то действия для профилирующей библиотеки retcode = PMPI_Xxx (...); какие-то действия для профилирующей библиотеки return retcode;

}

Отсюда видно, что сгенерировать профилирующую библиотеку нетрудно и можно сделать это автоматически.

Генератор профилирующей оболочки (wrappergen) создавался для дополнения профилирующего интерфейса MPI. Он позволяет пользователю написать любое число "meta" оболочек, которые могут быть применены к любому количеству функций MPI. Оболочки могут находиться в отдельных файлах и могут корректно вкладываться. Таким образом, на отдельных функциях может существовать более одного профилирующего слоя.

Генератор нуждается в трех источниках входной информации:

- список функций, для которых генерируется оболочка;
- объявления профилируемых функций. Для скорости и простоты разбора используется специальный формат (см. файл proto в каталоге mpich/mpe/profiling/wrappergen);
- определения оболочки.

Список функций есть просто файл с именами функций, разделенных пробельными символами. Если такого файла нет, любые foraiifn или fnaii макросы будут расширены до всех функций в декларационном файле (листинги П1.3—П1.5).

Листинг П1.3. Список функций для генератора библиотек

MPI_Abort

MPI_Address

MPI_Allgather

MPI_Allgatherv

MPI_Allreduce

MPI_Alltoall

MPI_Alltoallv

MPI_Attr_delete

MPI_Attr_get

MPI_Attr_put

MPI_Barrier

MPI_Bcast

MPI_Bsend

MPI_Bsend_init

MPI_Buffer_attach

MPI_Buffer_detach

MPI_Cancel

MPI_Cart_coords

MPI_Cart_create

MPI_Cart_get

MPI_Cart_map

MPI_Cart_rank

MPI_Cart_shift

MPI_Cart_sub

MPI_Cartdim_get

MPI_Comm_compare

MPI_Comm_create

MPI_Comm_dup

MPI_Comm free

MPI_Comm_group

MPI_Comm_rank

MPI_Comm_remote_group

MPI_Conim_remote_size

MPI_Comm_size

MPI_Coram_split

MPI_Comm_test_inter

MPI_Dims_create

MPI_Errhandler_create

MPI_E rrhandler_free

MPI_Errhandler_get

MPI_Errhandler_set

MPI_Error_class

MPI_Error_string

MPI_Finalize

MPI_Gather

MPI_Gatherv

MPI_Get_count

MPI_Get_elements

MPI_Get_processor_name

MPI_Graph_create

MPI_Graph_get

MPI_Graph_map

MPI_Graph_neighbors

MPI_Graph_neighbors_count

MPI__Graphdims_get

MPI_Group_compare

MPI_Group_difference

MPI_Group_excl

MPI_Group_free

MPI_Group_incl

MPI_Group_intersection

MPI_Group_range_excl

MPI_Group_range_incl

MPI_Group_rank

MPI_Group_size

MPI_Group__translate_ranks

MPI_Group_union

MPI_Ibsend

MPI_Init

MPI_Initialized

MPI_Intercomm_create

MPI_Intercomm_merge

MPI_Iprobe

MPI_Irecv

MPI_Irsend

MPI__Isend

MPI_Issend

MPI_Keyval_create

MPI_Keyval_free

MPI_Op_create

MPI_Op_free

MPI_Pack

MPI_Pack_size

MPI_Probe

MPI_Recv

MPI_Recv_init

MPI_Reduce

MPI_Reduce_scatter

MPI_Request_free

MPI_Rsend

MPI_Rsend_init

MPI_Scan

MPI_Scatter

MPI_Scatterv

MPI_Send

MPI_Send_init

MPI_Sendrecv

MPI_Sendrecv_replace

MPI_Ssend

MPI_Ssend_init

MPI_Start

MPI_Startall

MPI_Test

MPI_Test_cancelled

MPI_Testall

MPI_Testany

MPI_Testsome

MPI_Topo_test

MPI_Type_commit

MPI_Type_contiguous

MPI_Type_count

MPI_Type_extent

MPI_Type_free

MPI_Type_hindexed

MPI_Type_hvector

MPI_Type_indexed

MPI_Type_lb

MPI_Type_size

MPI_Type_struct

MPI_Type_ub

MPI_Type_vector

MPI_Unpack

MPI_Wait

MPI_Waitall

MPI_Waitany

MPI_Waitsome

MPI_Wtick

MPI_Wtime

Листинг П1.4. Фрагмент файла proto

128

int

MPI_Allgather

7

void *

sendbuf

int sendcount

MPI_Datatype sendtype

void * recvbuf

int recvcount

MPI_Datatype recvtype

MPI_Comm comm

int

MPI_Allgatherv

8

void *

sendbuf

int

sendcount

MPI_Datatype sendtype

void * recvbuf

int * recvcounts

int * displs

MPI_Datatype recvtype

MPI_Comm comm

int

MPI_Allreduce

6

void *

sendbuf

void * recvbuf

int count

MPI_Datatype datatype

MPI_Op op

MPI_Comm comm

int

MPI_Alltoall

7

void *

sendbuf

int sendcount

MPI_Datatype sendtype

void * recvbuf

int recvcnt

MPI_Datatype recvtype

MPI_Corran

comm

int

MPI_Alltoallv

9

void *

sendbuf

int * sendcnts

int * sdispls

MPI_Datatype sendtype

void * recvbuf

int * recvcnts

int * rdispls

MPI_Datatype recvtype

MPI_Comm

comm

int

MPI_Barrier 1 MPI_Comm

comm

int

MPI_Bcast 5 void *

Листинг П1.5. Описание оболочек для библиотеки трассировки MPE

#ifdef MPI_BUILD_PROFILING

#undef MPI_BUILD_PROFILING

#endif

#include <stdio.h>

#include "mpi.h"

{{fnail fn_name}} /*

{{fn_name}} — prototyping replacement for {{fn_naroe}}

Trace the beginning and ending of {{fn_name}}. */

{{vardecl int llrank}}

PMPI_Comm_rank(MPI_COMM_WORLD, &llrank);

printf("[%d] Starting {{fn_name}}...\n", llrank); fflush(stdout);

{{callfn}}

printf("[%d] Ending {{fn_name}}\n", llrank); fflush(stdout);

{{endfnall}}

Описание определений оболочки

Определения оболочки состоят из кода на языке C со специальными макросами. Каждое макроопределение окружается escape-последовательностью из двойных фигурных скобок {{ }}. Генератор может принимать следующие макроопределения:

- {{fileno}}

fileno — интегральный индекс, показывающий, какому файлу-оболочке принадлежит данный макрос. Это полезно при декларации глобальных переменных файла для предотвращения коллизий имен. Предполагается, что все идентификаторы, объявляемые вне функций, оканчиваются

_{{fiieno}}. Например:

static double overhead_time_{{fileno}};

можно расширить до:

static double overhead_time_0;

- {{forallfn <function name escape> <function A> <function B>...}} {{endforallfn}}

Код между {{forallfn}} и {{endforallfn}} копируется один раз для каждой профилируемой функции, исключая перечисленные функции, с заменой escape-строки, указанной в <function name escape>, именем каждой функции.

Например:

{{forallfn fn_name}(static int{{fn_name}}_ncalls_{{fileno}}; {{endforallfn}}

может быть расширено до:

static int MPI_Send_ncalls_l; static int MPI_Recv_ncalls_l; static int MPI_Bcast_ncalls_l;

- {{foreachfn <function name escape> <function A>...}}

{{endforeachfn}}

{{foreachfn}} — аналог {{forallfn}}, за исключением того, что оболочки пишутся только для функций, поименованных явно.

Например:

{{forallfn fn_name mpi_send mpi_recv}}

static int {{fn_name}}_ncalls_{{fileno}}; {{endorallfn}}

может быть расширено до:

static int MPI_Send_ncalls_2; static int MPI_Recv_ncalls_2;

- {{fnall <function name escape> <function A> <function B>... }}

{{callfn}} {{endfnall}}

{{fnall}} — определяет оболочку для использования во всех функциях, исключая названные. Генератор расширит инструкции до полного определения функции в традиционном для C формате. Макрос {{callfn}} укажет генератору оболочек, где вставлять вызов профилируемой функции. Должен быть только один экземпляр макроса {{caiifn}} в каждом определении оболочки. Макрос, указанный <function name escape>, будет замещен именем каждой функции. Внутри определения оболочки принимаются дополнительные макросы.

{{vardecl <type> <arg> <arg>... }}

vardeci — используется для объявления переменных внутри определения оболочки. Если вложенные макросы запрашивают переменные через vardecl с теми же самыми именами, генератор оболочек создаст уникальные имена добавлением

последовательных целых в конце запрашиваемого имени (var, vari, var2,...). Не рекомендуется объявлять переменные вручную в определении оболочки, т. к. объявления переменной могут появиться в коде позже, чем инструкции от других оболочек, что не соответствует классическому и ANSI C.

Если переменная объявлена через vardecl, требуемое имя для этой переменной (отличное от унифицированного имени, которое появится в окончательном коде) становится временным макроопределением, расширяемым к унифицированной форме. Например:

({vardecl int i d}} можно расширить до:

int i, d3; {{<argname>}}

Предлагаемое необязательное макроопределение, состоящее из имени одного из аргументов профилируемой функции, будет расширено до соответствующего аргумента. Это опциональное макроопределение служит, главным образом, только для утверждения, что профилируемая функция действительно имеет аргумент с данным именем.

{{<argnum>}}

На аргументы профилируемой функции можно ссылаться также и по номерам, начиная с нуля и двигаясь в сторону увеличения.

{{returnVal}}

ReturnVal расширяется до переменной, которая используется для хранения значения, возвращаемого профилируемой функцией.

{{caiifn}}

caiifn расширяется к вызову профилируемой функции. С определением вложенной оболочки оно также представляет точку, в которой вставляет-

ся код для любой вложенной функции. Вложения определяются порядком, в котором оболочки учитываются генератором. Например, если два файла, prof1.w и prof2.w, содержат две оболочки для MPi_Send, то профилирующий код, произведенный при использовании обоих файлов, будет иметь вид:

int MPI_Send(args. . .) arg declarations. . .

(

/*pre-callfn code from wrapper -1 from profl.w */

/*pre-callfn code from wrapper 2 from profl.w */

/*pre-callfn code from wrapper 1 from prof2.w */

/*pre-callfn code from wrapper 2 from prof2.w */

returnVal = MPI_Send(args . . .) ;

/*post-callfn code from wrapper 2 from prof2.w */

/*post-callfn code from wrapper 1 from prof 2. w */

/*post-callfn code from wrapper 2 from profl.w */

/*post-callfn code from wrapper 1 from profl.w */

return returnVal ;

- {{fn <function name escape> <f unction A> <f unction B>. . . }} {{callfn}}

{{endfnall}}

fn идентично fnall, за исключением того, что оно генерирует оболочки только для функций, именованных явно. Например:

{{fn this_fn MPI_Send}}

{ {vardecl int i}}

{{callfn}}

printf("Call to {{this_fn}} . \n") ;

printf ("{ {i} } was not used.\n);

printf ("The first argument to {{this_fn}} is {{0}}\n">;

{{endfn}}

будет расширено до:

int MPI_Send(buf , count, datatype, dest, tag, comm)

void * buf;

int count ;

```
MPI_Datatype datatype;

int dest;

int tag; MPI_Comm comm; {

int returnVal;

int i;

returnVal = PMPI_Send(buf, count, datatype, dest, tag, comm);

printf("Call to MPI_Send.\n");

printf("i was not used.\n");

printf("The first argument to MPI_Send is buf\n");

return returnVal;

}
```

Графические инструменты

В настоящее время графические инструменты MPE включают 3 программы: upshot для просмотра файлов в формате ALOG, Jumpshot-2 для файлов CLOG и Jumpshot-3 для SLOG-формата. Программа nupshot, представляющая собой улучшенный (более мощный) вариант upshot поставляется в дистрибутиве MPE, но не встраивается автоматически в общую структуру, т. к. использует старую версию Tcl/Tk, несовместимую с распространяемыми в настоящее время. Для сборки программ Jumpshot-2 нужен пакет JDK 1.1 с соответствующей версией Swing, его нельзя собрать с помощью JDK 1.2 или 1.3. Формат log-файла может быть изменен с помощью конвертеров формата, а logviewer избавит пользователя от необходимости помнить отношения между форматом log-файла и программой его просмотра.

Как уже отмечалось выше, для генерации log-файлов достаточно связать исполняемый модуль хотя бы с одной из профилирующих библиотек, а для MPICH просто указать одну из "профилирующих" опций компилятора:

```
mpif77 -mpilog -o fpilog fpilog.f
```

Полученный файл fpilog при исполнении будет генерировать log-файлы.

Команда

```
mpirun -np 4 fpilog
```

создаст файл с именем fpilog и расширением, зависящим от значения переменной Окружения LOG_FILE_FORMAT.

logviewer fpilog.slog

вызовет Jumpshot-3 для просмотра файла fpilog.slog.

Все графические инструменты, прилагаемые к MPE, представляют протоколы исполнения программ на каждом процессоре в виде последовательностей разноцветных прямоугольников, вытянутых вдоль прямой, соответствующей процессу. Расшифровка цветов и кнопки переключения режимов работы находятся на панели основного окна программы. Все программы имеют средства для детализации просмотра и выбора определенных операций.

Формат log-файла можно преобразовать с тем, чтобы использовать другой графический инструмент для его просмотра.

ПРИЛОЖЕНИЕ 2.

Средства отладки и мониторинга параллельных PVM-программ

Основным средством мониторинга и отладки параллельных PVM-программ является XPVM. XPVM — это гибрид графического монитора программ и графического интерфейса для PVM-консоли. Для того чтобы воспользоваться этим полезным средством, следует присвоить переменной окружения XPVM_ROOT значение — путь к исполняемому файлу xpvm. XPVM представляет собой относительно самостоятельный программный продукт, который может и не входить в дистрибутив PVM. В этом случае придется его устанавливать дополнительно к основному пакету. Кроме того, можно включить каталог с исполняемым файлом программы в путь поиска PATH.

При выполнении PVM-программы из XPVM генерируется трассировочная информация, которая записывается в специальный файл /tmp /xpvm.trace.USERNAME (USERNAME здесь — регистрационное имя пользователя). Этот файл может быть сохранен для последующего просмотра и анализа. XPVM предоставляет в распоряжение пользователя все возможности графического интерфейса. Это, прежде всего, доступ к командам обычной PVM-консоли через меню.

Запускается XPVM командой:

```
# xpvm
```

Если старт программы прошел успешно, загружается демон PVM (если виртуальная машина еще не создана, а если уже создана, она сохраняется вместе с выполняющимися на ней программами), а на экране появляется окно программы.

В конфигурационный файл $HOME/.xpvm_hosts может быть записана конфигурация виртуальной машины, загружаемая при запуске XPVM. Ключи запуска программы можно получить, введя команду:

```
# xpvm -HELP
```

(ключ -HELP должен вводиться в верхнем регистре).

В верхней части окна находится меню. Ниже основного меню располагается строка вывода кратких сообщений программы и окно просмотра конфигурации виртуальной машины. Затем идут строки с расшифровкой цветовой индикации процесса выполнения программы и кнопками изменения размера окон просмотра, а также кнопки проигрывателя трассировочных файлов. Здесь же содержится поле ввода имени трассировочного файла. Еще ниже находятся два окна, в левое выводится список задач выполняющейся программы, а в правое — пространственно-временная диаграмма параллельной программы. Под этими окнами расположены кнопки изменения размера окон и расшифровка цветовой индикации на пространственно-временной диаграмме. В нижней части находится строка вывода отладочной информации.

Основное меню программы содержит:

- пункт меню **File** содержит команды завершения работы с программой quit и halt. Они полностью аналогичны командам обычной PVM-кон-соли;
- пункт **Hosts** содержит команды добавления новых хостов в виртуальную машину и их удаления;
- в пункте **Tasks** содержатся команды управления программой. Это, прежде всего, команда запуска прикладной программы (пункт **spawn).** При нажатии кнопки, соответствующей этой команде, открывается диалоговое окно, в верхней части которого содержится поле ввода имени исполняемого файла программы. Имеются также поля, позволяющие задать количество запускаемых копий одной программы, ключи и маску отладки, хост или архитектуру для запуска главной задачи приложения и некоторые другие. В пункте **Tasks** имеются также команды посылки сигналов процессам;
- пункт **Views** позволяет включать и отключать окна просмотра XPVM. Чем больше таких окон включено, тем больше информации можно получить о процессе выполнения программы. С другой стороны, XPVM, как и всякая программа с графическим интерфейсом, потребляет значительные ресурсы -- процессорное время и память, поэтому наличие большого числа окон просмотра может заметно "тормозить" работу системы;
- пункт **Options** дает доступ к некоторым настройкам системы;
- команда **Reset** позволяет "сбросить" состояние виртуальной машины в исходное;
- назначение пункта **Help** очевидно — это доступ к встроенной справочной системе программы.

Окна просмотра

Четыре окна на панели XPVM служат для выдачи информации:

- о конфигурации виртуальной машины и использовании отдельных хостов;
- о загруженности системы;

- о ходе выполнения задания;
- о трассировке PVM-вызовов и программного вывода.

Окно просмотра конфигурации виртуальной машины

Окно просмотра конфигурации виртуальной машины (или сетевой конфигурации — Network View) содержит диаграмму, на которой прямоугольниками представлены-хосты, входящие в состав виртуальной машины. В каждом прямоугольнике находится название и пиктограмма операционной системы (пингвин, например, обозначает ОС Linux), сетевое имя компьютера. Прямыми линиями условно изображены коммуникации между компьютерами. В процессе выполнения параллельной программы прямоугольники окрашиваются в различные цвета, соответствующие разным состояниям процессов. Белый цвет обозначает отсутствие процесса PVM на данном хосте или пассивное состояние процесса, который не выполняет никакой работы. Желтый цвет соответствует обращению к подпрограмме PVM, а зеленый — выполнению "полезной" работы, например, вычислений. Если в процессе выполнения программы все или почти все прямоугольники окрашены в зеленый цвет, это означает равномерную загрузку хостов. Программа в этом случае хорошо сбалансирована, а эффективность ее выполнения близка к оптимальной. Если же часть хостов остается значительное время белой, они почти не участвуют в выполнении программы и реальная степень параллелизма программы меньше, чем, возможно, задумано программистом. Отображается и передача сообщений по линиям коммуникации.

Окно просмотра пространственно-временной диаграммы

Пространственно-временная диаграмма показывает состояние конкретных задач. Каждая задача изображается горизонтальным прямоугольником, отдельные части которого окрашены в разные цвета. Ось времени направлена по горизонтали, слева направо. В окне просмотра, находящемся слева от окна пространственно-временной диаграммы, для каждого прямоугольника выводится имя хоста и имя задачи. Зеленый цвет (Computing) показывает состояние выполнения вычислений или другой "полезной" работы. Желтый цвет (Overhead) соответствует вызовам подпрограмм PVM. Белый цвет (Waiting) обозначает простой процесса, состояние ожидания. Прямоугольники соединяются линиями, каждая из которых обозначает передачу сообщения. Если щелкнуть на линии левой кнопкой мыши, линия превращается в стрелку, которая показывает направление передачи сообщения, а в поле, находящемся ниже окна пространственно-временной диаграммы, выводится краткая информация о характеристиках сообщения (адресат, размер и т. д.). Вертикальная синяя линия показывает последний момент трассировки.

Диаграмма может оказаться слишком маленькой. Увеличить ее можно, нажав среднюю кнопку мыши (или левую и правую кнопки одновременно, если мышь двухкнопочная), и проведя по диаграмме. Уменьшить диаграмму можно, щелкнув в окне просмотра пространственно-временной диаграммы правой кнопкой мыши.

Окно просмотра степени использования системы

Окно просмотра степени использования системы содержит суммарную информацию о работе программы, всех ее процессов в каждый момент времени. Эта информация представлена тремя прямоугольниками, окрашенными в разные цвета. Нижний прямоугольник отображает долю работы программы, посвященную вычислениям (зеленый цвет), средний -вызовам подпрограмм PVM (желтый цвет) и верхний — простою программы (красный цвет). Относительная доля различных видов активности программы определяется по относительной высоте прямоугольников. Для хорошей программы в диаграмме должен преобладать зеленый цвет. Чем меньше красного, тем лучше.

Окна просмотра данных трассировки и программного вывода

Окна просмотра данных трассировки (Call Trace View) и вывода программы (Call Task Output View) можно использовать для отладки программы. Каждое событие трассировки представляет собой вызов подпрограммы PVM, сообщение о нем содержит значения параметров подпрограмм и коды завершения.

ПРИЛОЖЕНИЕ 3.

Настройка Linux-кластера для параллельных приложений

Введение

Кластерные вычислительные системы распространились так широко вследствие исключительной простоты своего устройства. С ростом скорости передачи данных по сети эффективность кластерных систем повышается, и это делает их еще привлекательнее. Небольшой вычислительный кластер из 6—10 персональных компьютеров может эффективно использоваться небольшим подразделением: отделом или лабораторией. Советы и рекомендации, приведенные в этом приложении, подразумевают настройку именно такой системы. Собрать и настроить кластер несложно, учитывая большое число публикаций на кластерные темы в Интернете. Как правило, это описание действий системных администраторов по настройке одного конкретного кластера. Действительно, обобщение накопленного опыта создания вычислительных кластеров и работы с ними составило бы не одну книгу, и за рубежом несколько таких книг уже выпущено. Возможно, они помогут найти ответы на вопросы: "Какие материнские платы нужны для вычислительных узлов?" или "Какие Ethernet-адаптеры мне следует покупать?". Но производители тем временем выпустят новые модели материнских плат и сетевых адаптеров, которые всегда лучше старых, и вопрос выбора оборудования по-прежнему будет связан с риском. Мы не будем, по возможности, рассматривать вопросы приобретения оборудования и ограничимся

рекомендациями общего характера по превращению группы PC в вычислительный кластер, опираясь на собственный опыт и обобщая материалы Интернета.

Общие положения

Будем считать, что кластер уже есть. В качестве минимального кластера можно принять двухпроцессорный сервер, на котором установлен MPICH, настроенный на работу через разделяемую память. Эта конфигурация уже позволяет экономить время за счет распараллеливания и, главное, дает принципиальную возможность для создания и отладки параллельных приложений. Но все-таки, это еще не группа компьютеров, объединенных сетью. Будем также считать, что на компьютерах кластера операционная система уже установлена. Какая? Желательно, одна из версий Linux RedHat, стабильная 6.1 или новейшая 7.3. Кластеры начинались с Linux, и когда потолком скоростей для Ethernet были 10 Мбит/с, именно под Linux писались драйверы для трех параллельных Ethernet-адаптеров, потому что именно так, используя вместо одной три карты Ethernet в одном компьютере, планировалось повысить скорость обмена по сети. Однако Linux, хоть и традиционная, но далеко не единственная кластерная операционная система. Есть кластеры, работающие под Windows NT, Free BSD, Solaris и т. д. Если вы по каким-то причинам выбрали одну из этих систем, наши советы по конфигурированию кластера будут вам не так полезны.

Далее, будем считать, что опыт администрирования UNIX у вас уже есть. Настроить кластер без умения администрировать невозможно. Помочь в этом читателю может книга, которая также вышла в издательстве "БХВ-Петербург": "Unix. Руководство системного администратора" Эви Немета, Гарта Снайдера, Скотта Сибасса и Трента Р. Нейна. Есть и другие книги, но эта, что называется, "поставит на ноги" и самого неопытного администратора.

И последнее условие: все компьютеры загружаются со своего собственного жесткого диска. Для отказа от бездисковой конфигурации с сетевой загрузкой нет уважительных причин: просто у нас нет такого опыта. На эту тему вы можете поискать материалы (а их много) в сети, например: http://www.scripps.edu/emetwall/ClusterHowto.html.

Чем кластер отличается от сети

Кластер нужно рассматривать как один многопроцессорный компьютер, не использующий- систему разделения времени и работающий в монопользовательском режиме — только тогда его работа будет эффективной. В каждом конкретном случае определяется свой подход к вопросам безопасности, прикладного программного обеспечения, администрирования, загрузки и использования файловой системы. Большинство дистрибутивов Linux способны поддерживать безопасность на очень высоком уровне. Однако в кластере эта безопасность, что называется, "идет в разрез" с вычислительными и административными установками.

Прикладное программное обеспечение использует системы передачи сообщений, подобные PVM или MPI. Есть и другие способы реализации параллелизма, но передача сообщений — самый популярный и приемлемый способ во многих случаях. Основным механизмом, запускающим параллельные процессы в узлах кластера, отличных от стартового, является беспарольный rsh для всех пользователей кластера.

Основная масса программного обеспечения создана в расчете на исполнение единственным CPU, в том числе и инструменты администрирования. Регулярная перегрузка вручную нескольких узлов вряд ли является лучшим способом хорошо провести время. К счастью, есть инструменты, помогающие администрировать кластер. Для поддержки возможностей эффективного администрирования приходится делать послабления в обеспечении безопасности внутри кластера. Очень полезно разрешить rlogin пользователю root для узла кластера, но чрезвычайно опасно для рабочей станции, находящейся в открытой сети.

Хотя мы решили не рассматривать бездисковый вариант, но необходимо отметить, что для загрузки с сервера также требуется несколько больше разрешений, чем для загрузки с собственного жесткого диска. И эти разрешения в открытой сети обычно не приветствуются.

И, наконец, файловые системы должны быть установлены так, чтобы, по крайней мере, каталог /home был доступен по NFS во всех узлах кластера. Разделение других файловых систем зависит от обстоятельств.

Общая установка

Из сказанного выше следует вывод: кластер должен существовать в закрытой (не анонсируемой) сети. Вот такая схема предлагается к реализации: кластер состоит из двух или более компьютеров под Linux, соединенных через коммутатор для Fast Ethernet (хаб тоже будет работать, но гораздо медленнее). Один узел назначается маршрутизатором и имеет выход в локальную сеть. Он же может выполнять функции "входного узла", который имеет клавиатуру, монитор и мышь. Остальным, так называемым вычислительным узлам клавиатуры, мониторы и мыши не нужны, хотя, конечно, они у них и могут быть.

Рис. П3.1 *иллюстрирует общую установку.*

Уровень подготовки

Установка и конфигурирование кластера требуют определенного навыка. В первом приближении потребуются: знакомство с конфигурированием сети, установки адресов, загрузки, конфигурирования ядра, драйверов Ethernet, NFS, NIS, установки пакетов и т. д. Документация к проекту LDP (Linux Documentation Project) превосходно изложена на http://metalab.unc.edu /LDP/.

Можете также взглянуть в "Beowulf HOWTO" на другие ссылки (http:// metalab.unc.edu/LDP/HOWTO/Beowulf-HOWTO-5.html#ss5.6).

Сетевое оборудование

Речь идет только об оборудовании, которое важно для приведения кластера в работоспособное состояние.

Зарубежные источники рекомендуют для лучшей производительности использовать Intel или DEC на базе tulip (2114X) сетевые Ethernet-адаптеры (http://cesdis.gsfc.nasa.gov/linux/drivers/tulip.html). Они обеспечивают "номинальную скорость проводов". Правда, удастся ли найти для них драйверы -это вопрос (http://maximus.bmen.tulane.edu/~siekas/tulip.html).

По нашему опыту, многие сетевые карты позволяют получить номинальную скорость сети. Единственное правило, которого нужно придерживаться неукоснительно, — это избегать приобретения неизвестных адаптеров от неизвестных производителей.

Адреса

Внутри кластера следует использовать зарезервированные IP-адреса. Эти адреса никогда не должны быть "видны" из Интернета. Используйте адреса в диапазоне 192.168.X.X или 10.X.X.X

Входной узел

Входной узел кластера должен действовать как gateway в локальную сеть. Этот узел должен иметь два сетевых соединения (и два Ethernet-адаптера), одно из которых получает адрес из не анонсируемой сети кластера, а другое имеет адрес из локальной подсети. Через входной узел можно осуществлять маршрутизацию, например, для подключения к файд-серверу.

Конфигурация коммутатора

Большинство коммутаторов идентифицируются назначенным им IP-адресом и администрируются через TELNET. Если в сети кластера используются адреса типа 192.168.0.X, то удобно назначить коммутатору адрес 192.168.0.254. При настройке переключателя следует полностью отключить все возможности спаннинга и настроить все порты на 100BTx-Full Duplex. Необходимо также проверить, что все карты и порты правильно выбирают скорости передачи.

Прямой доступ

В некоторых случаях может понадобиться непосредственный вход в тот или иной узел кластера через монитор и клавиатуру, главным образом, в случае возникновения проблем с сетью. Это можно сделать посредством дополнительного монитора и клавиатуры, подключаемых к нужному узлу. Другой путь — использовать KVM (keyboard-video-mouse), который позволяет работать с одним монитором, клавиатурой и мышью на всех компьютерах.

Решения вопросов безопасности

Рекомендуется, чтобы изменения в системе безопасности были сделаны только на вычислительных узлах, и ни в коем случае не делались на входном узле. Тогда уровень безопасности входного узла не пострадает:

- .rhosts против hosts.equiv. Есть два способа разрешить беспарольный rsh внутри кластера. Можно добавить запись в файл /etc/hosts.equiv, а можно завести файл .rhosts в домашнем каталоге каждого пользователя. Файл /etc/hosts.equiv содержится в недоступном для пользователей каталоге и поддерживается администратором. Формат файла .rhosts есть список компьютеров кластера.

Файл.rhost для кластера node:

права доступа 600

node1

node2

node3

node4

Формат файла /etc/hosts.equiv практически такой же.

#Файл hosts.equiv для кластера node

|название узла

nodel

node2

node3

node 4

Оба файла могут иметь дополнительные параметры, позволяющие устанавливать разрешения для беспарольного входа отдельным пользователям. Файл .rhosts, приведенный здесь, разрешает беспарольный вход с хостов nodel, node2, node3 и node4 тому пользователю, которому принадлежит. Файл /etc/hosts.equiv разрешает беспарольный вход всем действительным пользователям с компьютеров nodel, node2, node3 и node4. При разделении домашнего каталога по NFS файл .rhosts существует в единственном экземпляре и возлагает всю заботу о функционировании беспарольного rsh на своего владельца (который может использовать его не по назначению или некорректно). Файл /etc/hosts.equiv должен присутствовать на каждом узле кластера. Узлов может быть очень много, но сам файл находится в ведении системного администратора.

- rlogin для пользователя root. Просто добавьте файл .rhosts в домашний каталог пользователя root с правами на чтение и запись только для пользователя ("chmod go-rwx.rhosts").

Наконец, можно разрешить TELNET для пользователя root и FTP, которые обычно запрещены. Еще раз напомним, что все приведенные в этом приложении рекомендации относятся только к вычислительным узлам, они не допустимы для

входного узла.

Образец файла hosts

Приведем пример файла /etc/hosts для кластера из 4-х узлов и адресуемым коммутатором. Предполагается, что каждый узел получил соответствующий адрес при инсталляции или настройке сети:

#образец файла /etc/hosts

127.0.0.1 localhost localhost.cluster

192.168.0.1 nodel nodel.cluster

192.168.0.2 node2 node2.cluster

192.168.0.3 node3 node3.cluster

192.168.0.4 node4 node4.cluster

192.168.0.254 switch

Администрирование пользователей

Каждый пользователь должен быть зарегистрирован на каждом узле, который он хочет использовать. Проблем будет меньше, если каталог /home с одного из хостов (обычно, с входного узла) будет монтироваться по NFS. Обычной практикой при комплектации компьютеров кластера является отсутствие в них CD-ROM. Хорошо, если есть один CD-ROM, размещенный на входном узле.

Монтирование в кластере файловых систем входного узла. Нужно сделать так, чтобы все узлы монтировали каталог /home и CD-ROM с входного узла. Это значит, что собственный каталог /home на них не содержит ничего, и регистрировать пользователей на них не нужно. Убедитесь, что на каждом узле есть каталог /mnt/cdrom. Затем на каждом узле, кроме входного, добавьте в файл /etc/fstab записи:

hostnode:/home /home nfs bg,rw,intr 0 0 hostnode:/mnt/cdrom /mnt/cdrom nfs noauto,ro,soft 0 0

где hostnode: есть имя входного узла, разделяющего свои каталоги /home и /mnt/cdrom no NFS.

Следующее, что нужно сделать, — это изменить на входном узле файл /etc/exports так, чтобы он содержал такую запись:

#разрешаем компьютерам монтировать /home и /mnt/cdrom /home nodel(rw) node2(rw)

node3(rw)

/mnt/cdrom nodel(ro) node2(ro) node3(ro)

После этого необходимо перестартовать nfsd и mountd. Если все пойдет хорошо, то по запросу "mount /home" на любом вычислительном узле каталог /home будет смонтирован. Если возникнут проблемы, проверьте /var/log/messages и обратитесь к руководству по mount и nfs. П *Добавление пользователей.* Добавляйте регистрационные записи пользователей на входном узле так, как это делается на одиночной Linux-станции. Быстрый способ размножить регистрационные записи — это скопировать файлы /etc/passwd и /etc/shadow с входного узла на вычислительные. Убедитесь, что идентификаторы пользователей и групп на всех узлах совпадают. Теперь все пользователи могут входить на все узлы кластера. Но при любых изменениях в файлах — например, пользователь сменил пароль — придется повторять всю процедуру заново. Более правильный путь— установить NIS. Установка NIS является стандартной задачей для системного администратора, и мы отсылаем читателя к руководству по администрированию UNIX и страницам справочных пособий.

Администрирование CMS

Пакет CMS (Cluster Management System) можно найти по адресу http:// smile.cpe.ku.ac.th/software/scms/index.html. Он включает удаленный reboot и shutdown.

Компиляторы

Некоторые источники в Интернете рекомендуют использовать для компиляции программного кода передачи сообщений egcs (включая и g77) — см.: http://egcs.cygnus.com/. Версия: egcs-1.1.1. Стандартный gcc (не egcs gcc) предлагают использовать для генеральных задач, таких как компиляция ядра. Дело в том, что компилятор g77 является FORTRAN-компилятором на базе egcs.

Библиотеки передачи сообщений

Чтобы использовать группу компьютеров, соединенных сетью, как единую вычислительную систему, необходимо установить программное обеспечение передачи сообщений. Наиболее распространенными библиотеками передачи сообщений являются MPI и PVM.

- **MPI.** Есть две свободно распространяемые версии реализации протокола MPI (Message Passing Interface):
- **MPICH.** Источник: **http://www-unix.mcs.anl.gov/mpi/mpich/.** Название файла: mpich.tar.gz.
- **LAM-MPI.** Источник: **http://www.mpi.nd.edu/lam/.** Название файла: lam61 .tar.gz.

- **PVM.** Версия: pvm3/pvm3.4.beta7.tgz. Источник: **http://www.epm.ornl.gov /pvm/.**

Установка любого из этих пакетов не требует больших усилий. PVM и LAM-MPI входят в состав дистрибутива последних версий RedHat Linux, а для установки пакета из дистрибутива требуется только знакомство с командой rpm. (Например, для RedHat 7.3: pvm-3.4.4-2.i386.rpm, lam-6.5.6-4.1386.rpm.) MPICH устанавливается из архива исходных файлов и требует конфигурирования, компилирования и инсталляции. Это значит, что в процессе установки, в каталоге, образовавшемся при распаковке архивного файла с исходным кодом, необходимо выполнить скрипт ./configure с указанием параметров конфигурации. Для последних версий MPICH действительно достаточно выполнить конфигурацию без параметров (разве что указать параметр pref 1x=<имя *каталога для инсталляции MPICHУ)* — архитектура системы и тип устройства, используемого MPICH для кластеров PC (ch_p4) будут определены в процессе конфигурации.

Команды Make, Make test и Make install выполняют компиляцию пакета, тестирование и инсталляцию. Для настройки пакета нужно вписать все узлы кластера, с которыми планируется работать, в файл machines. LINUX. Файл располагается в подкаталоге share/ инсталляционного каталога и служит команде mpirun списком подчиненных хостов. Впрочем, mpirun имеет опцию

-machinesfile <*имя файла*>

по которой список используемых компьютеров может быть заменен. Чтобы сделать программы пакета MPICH доступными для пользования, нужно внести каталог с исполняемыми файлами, например, /usr/local/mpich /lib/LINUX/ch_p4, в переменную окружения PATH.

Установка PVM и LAM_MPI из исходных текстов не сложнее. PVM не требует никакой специфической настройки после инсталляции, кроме установки некоторых переменных окружающей среды для каждого пользователя. PVM при работе использует "виртуальную машину", конфигурируемую пользователем из pvm-консоли (консольного интерфейса) или из прикладных программ. Работу "виртуальной машины" поддерживает PVM-демон, запускаемый индивидуально для каждого пользователя. Обязательным условием успешной работы с PVM является корректное завершение работы всех PVM-демонов.

LAM-MPI также использует индивидуального пользовательского демона, обеспечивающего старт и окончание параллельных заданий, и отладку во время исполнения. Следует обратить внимание пользователей на корректное обращение с программой-демоном и обязательный останов демона командой "lamclean" по окончании работы.

Листинг П3.1. Назначение переменных окружения для PMV и LAM-MPI

#вставить в файл.bashrc для каждого пользователя #предполагается, что PVM и LAM-MPI установлены: export LAMHOME=/usr/local/lam61

export PVM_ROOT=/usr/local/pvm3

export PVM_ARCH=LINUX

export PATH=$LAMHOME:$PVM_ROOT/bin:$PATH

Тесты производительности

- Тест производительности сети: netperf.

Источник: **http://www.netperf.org/netperf/NetperfPage.html.**

- Тест производительности сети: netpipe.

Источник: **http://www.scl.ameslab.gov/Projects/Netpipe/.**

- Тесты производительности для параллельных систем: NASA parallel Benchmarks.

Источник: **http://www.nas.nasa.gov/NAS/NPB/.** П Тест производительности параллельных систем: Linpack. Источник: **http://www.netlib.org /benchmark/hpl/.**

Синхронизация часов

В любом случае будет лучше, если системные часы всех компьютеров кластера будут синхронизованы. Для синхронизации можно использовать ntp. Для этого нужно:

1. Установить все системные часы на текущее время.

2. Записать время в CMOS RTC командой clock -w.

3. На всех компьютерах установить пакет ntp (для RH7.3 ntp-4.1.1-U386.rpm).

4. Отредактировать, на каждом узле файл /etc/ntp.conf. На всех узлах закомментировать указанные строки:

faultleastclient

listen on default 224.0.1.1

#broadcastdelay 0.008

На вычислительных узлах (на всех, кроме входного) отредактировать строки:

server # local clock #fudge 127.127.1.0 stratum 0

5. На всех узлах запустить ntp-демона и включить его старт в стартовые скрипты.

Таким образом, мы синхронизировали все вычислительные узлы по локальным часам входного узла. Разумно ежесуточно записывать синхронизованное время в CMOS RTC, поставив в список сгоп для пользователя root команду

/sbin/clock -w

Вот тот минимальный набор заданий, необходимых для организации вычислительного кластера. Кластер такой конфигурации будет удобен для небольшой (15—20 человек) группы постоянных пользователей, имеющих право консольного входа на входном узле. Там, где число пользователей превышает несколько десятков человек, или их работа с кластером организуется через локальную сеть или модемный пул, необходимо устанавливать программы менеджирования пользовательских заданий — системы очередей и организовывать работу пользователей с кластером через Web-интерфейс.

ПРИЛОЖЕНИЕ 4.

Ресурсы Интернета, посвященные параллельному программированию

Как найти в Интернете информацию о высокопроизводительных вычислениях? Интернет безграничен и проторенных высокопроизводительных путей в нем, на первый взгляд, нет. Поисковые системы? Самая популярная поисковая система в мире **http://www.google.org,** на запрос "Parallel Computing" за 0,25 секунды находит около 1 080 000 подходящих ссылок. Наш российский Яндекс **(http://www.yandex.ru)** "отстает", выдавая "только" 3207 страниц, среди которых опытный глаз сразу ловит знакомые ссылки. А как быть неопытному?

Если информация, приведенная в нашей книге покажется вам недостаточной, попробуйте найти нужные источники на серверах, предоставляющих информацию в структурированном (наподобие дерева) виде. Наиболее удачный вариант **http://dmoz.org.** Путь Top:Computers:Parallel Computing: Pro-gramming:Libraries от главной страницы сайта выводит к необходимому списку: BSP, MPI, PVM. Правда, в каждом из этих разделов по несколько десятков ссылок и ссылки добавляются добровольцами, следовательно, неравноценны. Но все-таки главного вы не пропустите. По такому принципу устроены многие российские информационные сайты, например, **http://www.pingwin.ru,** только информации о параллельных и высокопроизводительных вычислениях на них нет.

Практически все крупные университеты США имеют свои суперкомпьютерные центры и связанную с высокопроизводительными вычислениями перспективную программу развития. Каждый из этих центров располагает уникальным набором суперкомпьютеров и обширной информационно-справочной системой, направленной на поддержку своих пользователей.

Вершина мирового суперкомпьютинга: **http://www.top500.org.** Центральный элемент сайта — таблица производительности суперкомпьютеров по тесту Linpack (ссылки на тестирующую программу и рекомендации по ее применению прилагаются). В таблицу входят 500 наиболее производительных систем в мире. Если хотите быть в курсе всех технических новинок в области процессоростроения, новостей из жизни мировых лидеров разработки и производства высокопроизводительных систем и знать, как идут дела в самом новом суперкомпьютерном суперпроекте, да к тому же свободно

читаете по-английски, то назначьте этот сайт домашней страничкой своего интернет-обозревателя.

Если же у вас с английским проблемы, то поищите "вершину" поближе. Это **http://parallel.ru,** ведущий информационный сайт российского суперкомпьютинга. На нем собраны материалы по всем аспектам высокопроизводительных вычислительных технологий. Информация хорошо структурирована и регулярно обновляется. Там же можно включить себя в список рассылки **par-news@parallel.ru** и получать по электронной почте дайджест новостей российского и мирового суперкомпьютинга на русском языке с указанием источника информации.

Сайты лидеров Санкт-Петербургского суперкомпьютинга:

- **http://www.csa.ru** — Институт высокопроизводительных вычислений и баз данных;
- **http://www.ptc.spbu.ru** — Петродворцовый телекоммуникационный центр (Санкт-Петербургский государственный университет) и его суперкомпьютерный центр;
- **http://www.hpc.nw.ru** и **http://www.hi-hpc.nw.ru** — сайты, посвященные высокопроизводительным вычислениям, Санкт-Петербургский государственный университет. Здесь, в частности, можно получить доступ к дистанционным курсам по методам и технологиям программирования для высокопроизводительных вычислительных систем.

Многолетний опыт работы по пропаганде и распространению технологий высокопроизводительных вычислений нашел отражение на персональной странице одного из авторов: **http://mph.phys.spbu.ru:8080/~nemnugin/.**

Информация о MPI

Где можно найти информацию о MPI? Есть спецификация Message Passing Interface (MPI), и есть его реализация — программный пакет MPICH. Соответственно, есть два разных адреса: **http://www.mpi-forum.org** (не путать с **http://www.mpi.org)** и **http://www-unix.mcs.anl.gov/mpi/mpich/.** На первом сайте размещена информация и архивы, связанные с процессом разработки стандарта передачи сообщений. На втором сайте вы найдете рекомендации по установке пакета MPICH, а также сможете загрузить оттуда по ftp последнюю версию пакета (на момент написания книги это версия 1.2.4).

Другие реализации протокола MPI:

- **http://www.lam-mpi.org** - LAM (Local Area Microcomputer) реализация MPI для разнородных компьютерных кластеров;
- **http://www-csag.ucsd.edu/projects/comm/mpi-fm.html** - MPI-FM, "переложение" пакета MPICH для высокопроизводительных кластеров на интерфейс Fast Messages — низкоуровневый протокол быстрой передачи сообщений;

- **http://www.lfbs.rwth-aachen.de/users/joachim/MP-MPICH/ - MP-MPICH,** многоплатформенный MPICH. Это расширение MPICH для Windows NT/2000, приспособленное к работе через разделяемую память, интерфейс SCI и TCP/IP;
- **http://www.scali.com/Products/scampi.shtml** - небесплатная (зато самая быстрая) реализация MPI через интерфейс SCI;
- **http://www.criticalsoftware.com/wmpi/** — реализация MPI для Microsoft Windows;
- **http://www.ens-lyon.fr/LIP/RESAM/** — MPI для Myrinet.

Информация о PVM

Где можно найти информацию о PVM? Официальная страница комплекса Parallel Virtual Machine находится по адресу **http://www.epm.ornl.gov/pvm/.** Там вы найдете ссылки на учебники, сообщения об ошибках в PVM и другие полезные ссылки. Загрузить дистрибутивы для Linux и Windows можно с **http://www.netlib.org/pvm3/.** Кроме того, PVM входит в виде архива rpm в дистрибутив Linux RedHat, правильно разворачивается и не требует дополнительных настроек.

Другие версии и расширения пакета PVM:

- **http://www.epm.ornl.gov/pvm/NTport.html** — PVM для MS Windows;
- **ftp://ftp.cs.umn.edu/users/du/pvm-atm/www.html** — PVM через ATM;
- **http://atc.ugr.es/javier-bin/pvmtb_eng** — инструментарий PVM для математического пакета Matlab;
- **http://citeseer.nj.nec.com/zhou971pvm.html -** экспериментальная реализация PVM для систем с разделяемой памятью, через подпроцессы (threads);
- **http://theoryx5.uwinnipeg.ca/CPAN/data/Pvm/blib/Pvm.html** - Perl-вариант PVM.

Информация о кластерах

Где можно найти информацию о кластерах? Самый "суперкластерный" сайт в мире — это **http://www.beowulf.org.** История развития проекта "Beowulf, драйверы, технические советы и рекомендации, адреса кластерных сайтов, статья "How to build a Beowulf. Если вы хотите сделать свой собственный суперкомпьютер кластерного типа, то информация от Beowulf вам пригодится. Если хотите быть в курсе кластерной суперпроизводительности, не пропустите адрес **http://www.topclusters.org/.** Самый практичный, с точки зрения скромного российского "кластеростроителя", сайт — это COCOA, **http://cocoa.aero.psu.edu/.** Если 25 двухпроцессорных рабочих станций фирмы DELL для ваших замыслов недостаточно, обратите внимание на **http://www.lsc-group.phys.uwm.edu/beowulf/.** Этот сайт посвящен кластеру из 300 узлов DEC Alpha в университете штата Висконсин.

Разумеется, приведенные здесь ссылки далеко не исчерпывают доступные через Интернет ресурсы по высокопроизводительным вычислениям. Пользуйтесь

поисковыми машинами, заходите на сайты авторов, где регулярно обновляются полезные ссылки.

Список литературы

1. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing — The MIT Press, Cambridge, 1994.
2. A. K. Ewing, H. Richardson, A. D. Simpson, R. Kulkarni. Writing Data Parallel Programs with High Performance FORTRAN — Edinburgh Parallel Computing Centre, The University of Edinburgh.
3. A. Marshall. High Performance Fortran — интернет-курс **http://www.psu.edu /dept/cac/ait/nic_group/Edu_Train/HPF_tutorial/,** Liverpool, 1996.
4. N. MacDonald, E. Minty, T. Harding, S. Brown. Writing Message Passing Parallel Programs with MPI -- Edinburgh Parallel Computing Centre, The University of Edinburgh.
5. Водяхо А. И., Горнец Н. Н., Пузанков Д. В. Высокопроизводительные системы обработки данных. — М.: Высшая школа, 1997.
6. Меткалф М., Рид Дж. Описание языка программирования Фортран 90. -М.: Мир, 1995.
7. Немнюгин С., Чаунин М., Комолкин А. Эффективная работа: UNIX. -СПб.: Питер, 2001.