МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ "СТАНКИН"

Учебно-методическое объединение по образованию в области автоматизированного машиностроения (УМО АМ)

М.В. Якобовский

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ И СЕТИ

Рекомендовано Учебно-методическим объединением по образованию в области автоматизированного машиностроения (УМО АМ) в качестве учебного пособия для магистров, обучающихся по направлениям: "Технология, оборудование и автоматизация машиностроительных производств"; "Автоматизация и управление" вузов объединения.

Москва 2000

УДК 681.324.012:519.6
Я 46

Рецензенты:
д.ф.-м.н., проф., Серебряков В.А.
к.ф.-м.н., с.н.с., Поляков С.В.

Я 46 Якобовский М.В. Распределенные системы и сети. Учебное пособие. - М.: МГТУ "Станкин", 2000.- 118с., ил.

В учебном пособии изложен материал, охватывающий следующие разделы: 1. Архитектура многопроцессорных систем. 2. Принципы построения параллельных алгоритмов. 3. Синхронизация последовательных процессов. 4. Языки и средства параллельного программирования. 5. Параллельное программирование в системе "PARIX™". 6. Проблема балансировки загрузки. 7. Визуализация в распределенных вычислительных системах. В пособии содержится так же ряд приложений, которые могут быть использованы при написании программ.

Учебное пособие предназначено для магистров, обучающихся по направлениям 550200, 552800, 552900.

Рис. 61. Табл. 6. Библиогр. 39 наз.

Содержание

Распределенные системы и сети

Введение

Одним из наиболее интересных направлений развития современной вычислительной техники являются многопроцессорные системы. Толчком к их массовому распространению послужил выпуск в 1985г. первого транспьютера – разработанной английской фирмой INMOS сверхбольшая интегральная схема, объединяющей в одном корпусе процессор и четыре последовательные канала межпроцессорной передачи данных. Характерный представитель этого семейства – транспьютер IMS T800 обладает скромной, по современным стандартам, производительностью порядка 1 миллиона операций в секунду. Несмотря на это, именно транспьютеры предоставили возможность построения на их основе относительно дешевых систем с практически неограниченным числом вычислительных узлов, объединенных для совместного решения задач самого разного плана. Даже когда транспьютеры перестали представлять сколько-нибудь серьезный интерес в качестве вычислительных элементов, системы, построенные на основе гибридных процессорных узлов, включающих высокопроизводительный процессор, например i860 или PowerPC, в качестве вычислительного компонента и транспьютер - в качестве коммуникационного процессора, на значительный срок определили облик многопроцессорных систем массового параллелизма. Их заслуга еще и в том, что они стимулировали активную работу большого числа ведущих научных коллективов мира в плане практического создания параллельных алгоритмов и программ для решения самого широкого круга задач. Отдавая дань той роли, которую транспьютеры сыграли в плане развития многопроцессорной техники системы с распределенной памятью часто называют транспьютероподобными.

Применение параллельных систем

Коротко охарактеризуем основные возможности, предоставляемые многопроцессорными системами, и стимулирующие разработку и использование распределенных систем.

- ***Решение задач, недоступных для выполнения на однопроцессорных системах.*** – В качестве примера приведем задачу моделирования климата в планетарном масштабе. Для ее решения требуется оперативная память, измеряемая терабайтами и производительность, измеряемая терафлопами.
- ***Решение задач реального времени.*** – Характерными представителями задач этого класса являются проблемы управления движением самолетов в воздушной зоне аэропорта, сканирования качества поверхности стального листа в ходе его проката, тре-

бующие распознавания особенностей ситуации и принятия адекватного решения за короткое, наперед заданное время.

- ***Сокращение времени решения задач.*** - Характерный пример - подготовка прогноза погоды на следующий день. Его точность напрямую зависит от производительности используемой вычислительной системы, поскольку время, за которое он должен быть подготовлен, вполне определено – завтрашний прогноз должен быть готов сегодня.
- ***Построение систем высокой надежности.*** - Возникновение однократных ошибок является нормой функционирования бортовой техники. Поэтому необходимо уметь автоматически распознавать и исправлять эти ошибки в режиме реального времени.
- ***Одновременное обслуживание множества потоков данных***. Сбор информации со станций мониторинга экологической, сейсмической, метеорологической или иной обстановки, разбросанных по обширной территории, или с датчиков, установленных вдоль технологической линии, как правило, невозможно производить с помощью единственного компьютера.
- ***Информационное обслуживание распределенных баз данных***. - Поиск информации в распределенной компьютерной сети требует привлечения ресурсов ее узлов, превращая ее таким образом в подобие многопроцессорной системы на период выполнения запроса.

Укажем основные направления использования многопроцессорных систем.

- **Задачи «большого вызова» (Grand Challenge)**

Многопроцессорные системы с успехом используются для изучения задач, решение которых невозможно на вычислительных системах традиционной архитектуры сегодняшнего дня. В первую очередь, это задачи глобального моделирования климата, предсказания погоды, мониторинга состояния воздушного и водного бассейнов Земли, моделирования экологических проблем, моделирование режимов горения топлива в котлах и двигателях внутреннего сгорания, горения метана при авариях на скважинах и трубопроводах. Сюда же относятся проблемы моделирования сверхзвукового обтекания летательных аппаратов сложной формы, моделирования с высокой точностью полупроводниковых элементов сверхмалых размеров, составляющих основу современной и перспективной компьютерной техники и многие другие. Экспериментальное исследование подобных проблем либо не представляется возможным, либо обходится чрезвычайно дорого, тогда как проведение соответствующих широкомасштабных вычислительных экспериментов открывает реальные возможности интенсив-

ного развития указанных и ряда других областей. Однопроцессорные системы не обладают соответствующими ресурсами оперативной памяти и быстродействием, тогда как масштабируемые многопроцессорные системы предоставляют соответствующие возможности.

Рассмотрим только один пример. Сетка, покрывающая атмосферу на высоту до 50 км, с разрешением 1 точка на км^3 содержит порядка 6 миллиардов узлов. Общий объем вычислительной работы при моделировании одного месяца с шагом 1 мин в предположении, что на одну точку при переходе от одного временного слоя к другому тратится всего 100 операций, составит порядка 10 миллионов миллиардов операций. Системы, занимающие первые десять позиций списка 500 крупнейших суперкомпьютеров мира, в принципе в состоянии выполнить эту работу за 1 день, что позволяет говорить о реальной возможности решения таких задач уже сегодня. Эти системы содержат в своем составе от сотен до тысяч процессоров (первую позицию сегодня занимает крупнейшая система ASCI Red - 3 Sandia National Labs, USA, объединяющая 9.472 процессора пиковой производительностью 3.154 триллионов операций с плавающей точкой в секунду - www.top500.org).

- **Решение задач реального времени**

Характерным примером задачи реального времени является обработка потока кадров двумерных изображений. Однородный характер действий, выполняемых над растровой разверткой кадра, позволяет без особых затруднений использовать большое количество процессоров. Объединенные в решетки транспьютеры успешно используются для быстрой обработки потоков кадров, например при спутниковой фотосъемке земной поверхности или при распознавании банкнот. Одна из первых немеханических машин была разработана Г.Холлеритом для анализа результатов переписи населения в США в 1890г. Несколько лет назад 32-процессорная система PowerXplorer успешно использовалась для подведения итогов голосования при выборах в ФРГ. С ее помощью автоматически анализировались и учитывались бюллетени избирателей. Известны эффективные системы анализа качества стального прокатного листа, основанные на быстром контроле его поверхности с помощью установленных над конвейерной линией видеокамер.

Транспьютероподобные системы эффективно используются в качестве управляющих элементов в технологических линиях, различных компьютерных тренажерах и робототехнике, в телекоммуникационных системах.

- **Проектирование систем высокой надежности**

Транспьютеры, допускающие построение на их основе вычислительных систем с избыточным числом процессорных узлов, идеально подходят для конструирование бортовых компьютерных систем космической и подобной техники, требующей повышенной надежности функционирования. Популярности именно транспьютеров для использования в подобных системах немало способствовали высокие технологические качества самих транспьютерных элементов, их высокая отказоустойчивость, возможность построения на их основе систем реального времени с гарантированным временем отклика.

- **Создание информационной инфраструктуры современного общества**

Хорошо известным представителем распределенной вычислительной системы является глобальная сеть Internet, объединяющая сотни миллионов компьютеров всего мира в единое информационное пространство. Построенная на несколько иных принципах, нежели транспьютерные системы, она, тем не менее, имеет с ними много общего. С ее помощью можно не только эффективно решать задачи поиска разнообразной информации в огромной распределенной, динамически изменяющейся базе данных. Известны примеры решения трудоемких задач, например поиска простых делителей больших чисел на ряде компьютеров, разбросанных по всему миру. Соответствующие программы использовали машинное время в периоды простоя удаленных вычислительных систем, приостанавливаясь при появлении на них активных заданий локальных пользователей.

Коммуникационные сети

С момента появления систем массового параллелизма идет борьба за построение сбалансированных систем, в которых пропускная способность каналов межпроцессорного обмена данными соответствует производительности вычислительных узлов. К тому моменту, когда производительность вычислительных процессоров превысила 100 Mflops (миллионов операций с плавающей точкой в секунду), стало очевидно, что транспьютерные каналы, поддерживающие передачу данных со скоростью примерно 1 Мбайт/с, уже не удовлетворяют современным требованиям, и разными фирмами были предложен ряд решений, обеспечивающих бóльшую пропускную способность. Здесь можно отметить сети Fast Ethernet, обеспечивающие на сегодня скорость передачи данных порядка 9 Мбайт/с, получившие широкое распространение сети Myrinet фирмы MYRICOM (~ 20 Мбайт/с), специализированные сети, например, на основе концентраторов и адаптеров HS-link германской фирмы Parsytec (~ 40 Мбайт/с). Отметим, что приведенные характеристики не отражают полностью возможности ука-

занных сетей. Они в значительной мере зависят как от аппаратных особенностей использующих их вычислительных систем, так и от типа используемых протоколов передачи данных. На сегодня ведутся работы по построению масштабируемых сетей с пропускной способностью 200 Мбайт/с - 1Гбайт/с и выше.

Но, что особенно важно, несмотря на некоторое изменение способов объединения процессорных узлов между собой и среды передачи данных, для разработчика прикладного обеспечения ситуация изменилась незначительно. Можно с уверенностью говорить о совместимости, с точки зрения созданных алгоритмов и методов, систем самого широкого круга, что обусловлено общностью принципов построения систем с распределенной памятью.

Отрадным примером внимательного отношения к проблемам переносимости прикладного программного обеспечения, с одной стороны, и эффективного использования аппаратных возможностей вычислительных систем, с другой, является оригинальная разработка немецкой фирмы Parsytec – система программирования PARIX™, сводящая к минимуму расходы на адаптацию прикладного обеспечения при его переносе между многопроцессорными системами разных серий этой фирмы.

Суперкомпьютеры - вызов мировому научному сообществу

Появление в последнее время многопроцессорных систем терафлопной производительности, объединяющих тысячи процессоров, представляет определенный вызов нашим способностям эффективно использовать предоставляемые ими возможности. Обладая впечатляющей вычислительной мощностью, большая часть современных мультипроцессорных суперкомпьютеров оказалась востребованной всего на несколько процентов. Иначе говоря, 90% времени многие такие системы простаивают.

Быстрое развитие многопроцессорной техники привело к возникновению ряда проблем, нетипичных для вычислительных систем традиционной архитектуры. Речь идет о факторах, не связанных непосредственно с проблематикой решаемых на суперкомпьютерах задач, но значительно затрудняющих использование систем массового параллелизма.

На первом месте традиционно отмечают сложность адаптации последовательных алгоритмов к параллельным системам вообще и к системам с распределенной памятью, в особенности. Значительная доля хорошо зарекомендовавших себя алгоритмов не имеет эффективных параллельных аналогов. В связи с этим основные силы на протяжении длительного времени сконцентрированы на поиске и изучении новых параллельных алгоритмов. Этим обусловлен определенный

прогресс в области построения параллельных алгоритмов для решения широкого спектра прикладных задач.

В качестве следующего препятствия на пути эффективного использования параллельной техники справедливо называют проблемы обеспечения равномерной загрузки процессоров в ходе решения задачи. В этом направлении также ведутся интенсивные исследования, известен широкий спектр алгоритмов как статической, так и динамической балансировки загрузки, не только обеспечивающих, в зависимости от задачи, более или менее равномерное распределение вычислительной нагрузки между процессорами, но и минимизирующих объем передаваемых между ними данных. Однако, несмотря на ощутимые успехи в решении отмеченных проблем, типичный коэффициент интегральной загрузки большинства систем массового параллелизма составляет по разным оценкам всего от 10% до 3%. И это несмотря на наличие алгоритмов и программ, обеспечивающих весьма высокий (часто более 90%) коэффициент распараллеливания при решении ряда задач.

При переходе на технику массового параллелизма значительно возрастает нагрузка на разработчика прикладного обеспечения, причем ситуация усугубляется отсутствием в новых системах привычных инструментов разработки программ (удобных отладчиков, интегрированных сред программирования и т.д.) и анализа получаемых результатов. Аналогичные сложности испытывает и системный администратор, отвечающий за функционирование комплекса. Первое приводит к значительным затратам на адаптацию уже не алгоритмов, но коллектива разработчиков к новой ситуации - процесс, занимающий не один месяц и даже не один год. Второе приводит к многочисленным простоям вычислительной системы даже в условиях наличия достаточного набора задач, способных в принципе обеспечить близкую к 100% загрузку системы, так как служба системного администратора просто не в состоянии обеспечить эффективную дисциплину запуска задач без соответствующего системного обеспечения, а оно отсутствует практически всегда. Должно пройти заметное время, прежде чем многопроцессорные системы станут стандартом де-факто и появятся надежные распределенные операционные системы с адекватными и удобными средствами управления.

Немаловажным фактором, снижающим эффективность использования мощных высокопроизводительных вычислительных систем, являются проблемы хранения, обработки и визуализации больших объемов данных, неизбежно сопровождающих проводимые масштабные вычислительные эксперименты. Так или иначе, визуализацией данных вынуждены заниматься практически во всех исследовательских центрах, обладающих техникой нетрадиционной архитектуры.

Параллельная обработка

Основную идею, позволяющую нам использовать группу процессоров для совместного решения некоторой задачи, можно пояснить на примере завтрака английского джентельмена. К завтраку готовят чашечку кофе, хот-дог и салат, предваряет завтрак традиционное чтение утренней газеты. Процесс приготовления завтрака и чтения газеты джентльменом представлен на рис. 1. Длина рамок, заключающих в себе названия этапов, пропорциональна времени выполнения соответствующего этапа, стрелка в нижней части рисунка соответствует общему времени приготовления завтрака и чтения газеты.

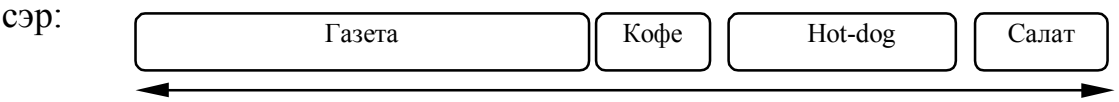

Рис. 1. Последовательное приготовление завтрака

Очевидно, что рассмотренный процесс последовательного приготовления кофе, бутерброда, салата и чтения газеты таит определенные резервы с точки зрения распределения работы между членами семьи джентльмена. Можно было бы ожидать 4-кратного, по числу этапов, сокращения общего времени выполнения всей необходимой работы. Рассмотрим ряд вариантов распределения работы. Первый вариант, представленный на рис. 2, позволяет сократить общее время примерно вдвое.

Рис. 2. Параллельное приготовление завтрака, высокая занятость работников

Джентльмен дочитает газету раньше, чем жена закончит приготовление хот-дога. Можно несколько ускорить процесс за счет привлечения к процессу третьего работника. Соответствующее решение представлено на рис. 3.

Обратим внимание на то, что время приготовления завтрака тремя людьми, хотя и сократилось по сравнению со вторым вариантом, но по сравнению с первоначальным вариантом сократилось отнюдь не в три раза. Причина в том, что значительную

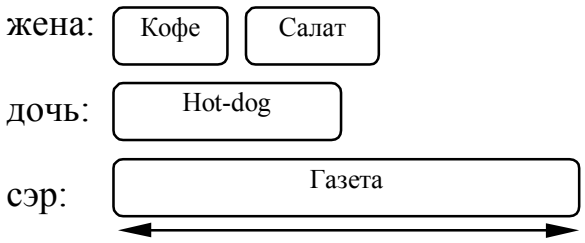

Рис. 3. Параллельное приготовление завтрака, наиболее быстрый метод, средняя занятость работников

часть времени, пока выполняется наиболее длительное действие – чтение газеты, жена и дочь не выполняют никакой полезной, с точки зрения нашей задачи, работы. Хорошо видно, что общее время выполнения определяется наиболее длительным этапом. В рассматриваемой задаче этот этап не удается разбить на части и обеспечить участников работой более равномерно. Точно так же не ускоряет процесса дальнейшее увеличение числа работников, хотя задание допускает одновременную работу на первоначальном этапе не 3-х, а 4-х человек (рис. 4).

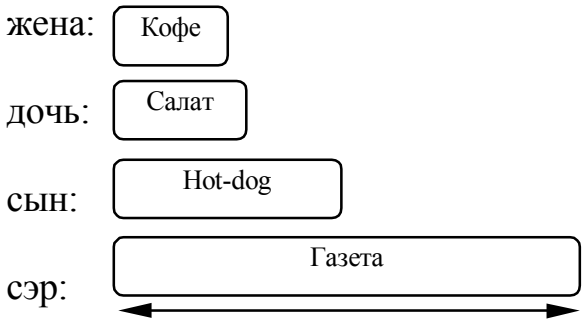

Рис. 4. Параллельное приготовление завтрака, неэффективное распределение работы

Из приведенного примера можно сделать некоторые предварительные выводы:

- использование для решения задачи ***p*** обрабатывающих элементов приводит, в общем случае, к сокращению времени ее решения менее чем в ***p*** раз;
- начиная с некоторого момента дальнейшее увеличение числа обрабатывающих элементов не приводит к сокращению времени решения задачи.

В дальнейшем мы убедимся в их справедливости. Будет показано, что возможности сокращения времени решения задачи на многопроцессорной системе в первую очередь определяются внутренними свойствами задачи и алгоритма, выбранного для ее решения. Значительно сократить время выполнения алгоритма, не обладающего внутренним параллелизмом, не удастся, какой бы вычислительной системой мы ни воспользовались.

Тема 1. Архитектура многопроцессорных систем

Среди всего разнообразия многопроцессорных систем рассмотрим системы типа МКМД (**М**ножественные потоки **К**оманд и **М**ножественные потоки **Д**анных). Вычислительные системы типа МКМД, представляют собой объединение процессоров, на каждом из которых может быть запущена своя, отличная от других программа, оперирующая своими собственными данными. В свою очередь МКМД системы можно разделить на два класса – системы с общей памятью и системы с раздельной памятью.

Системы с общей памятью

С точки зрения программиста привлекательно выглядят системы с общей памятью. Разбитая на взаимодействующие процессы (нити) программа в большинстве таких систем автоматически распределяется по доступным процессорам системы. Для систем этого класса существуют средства автоматизированного построения параллельных программ - векторизующие компиляторы. Для определенного круга задач они дают хорошие результаты в плане эффективности создаваемого программного кода. Достаточно широкий круг последовательных алгоритмов может быть успешно адаптирован к функционированию на системах с общей, или, что то же самое, разделяемой памятью (рис. 5). С точки зрения системного администратора, система с общей памятью привлекательна уже хотя бы потому, что функционирует под единственной копией операционной системы и не требует индивидуальной настройки каждого процессорного узла. Подобные компьютеры, управляемые UNIX-подобными операционными системами, обслуживают пользователей в режиме коллективного пользования, автоматически распределяя поток прикладных приложений по процессорам без участия администратора или оператора.

Рис. 5. Многопроцессорная система с общей памятью

Однако у систем с общей памятью есть ряд существенных недостатков:

- Относительно небольшое число процессоров;
- Отсутствие возможности наращивания числа процессоров - масштабируемости;
- Пиковая производительность крупнейших систем с общей памятью ниже пиковой производительности крупнейших систем с раздельной памятью;

- Высокая, относительно аналогичных по производительности систем с раздельной памятью, стоимость.

На сегодняшнем уровне технического развития эффективное объединение более 20 - 30 процессоров на основе общей памяти затруднительно. Каждый процессор должен иметь физический доступ к каждому из блоков оперативной памяти. Например при использовании раздельных 32-разрядной адресной шины и 64-разрядной шины данных требуется минимум 96 разрядная высокоскоростная линия доступа к памяти от каждого процессора, что само по себе уже представляет технологическую проблему. Необходимость обеспечения обмена со скоростью 500 Мбайт/с и выше ограничивает физическое расстояние от каждого процессора до каждого блока памяти. Как правило, системы с общей памятью действительно выглядят достаточно компактно, размещаясь в одном корпусе. Этот фактор ограничивает общее число процессоров числом блоков, которые можно разместить в пределах, заданных максимальным расстоянием от блока памяти до процессора. Но есть и более существенные ограничения. В принципе, общая память предполагает возможность всех процессоров одновременно прочесть или записать разные данные из одного и того же блока памяти. Для этого каждый блок памяти должен обладать числом точек входа, равных числу процессоров, что технически сегодня нереализуемо. В качестве примера возможного решения этой проблемы приведем схему сети типа "бабочка" (рис. 6). Согласно этой схеме 8 процессорных модулей могут получить доступ к восьми блокам памяти через систему коммутаторов, однако в каждый конкретный момент к каждому блоку памяти может иметь доступ только один из процессоров, что в целом снижает производительность системы.

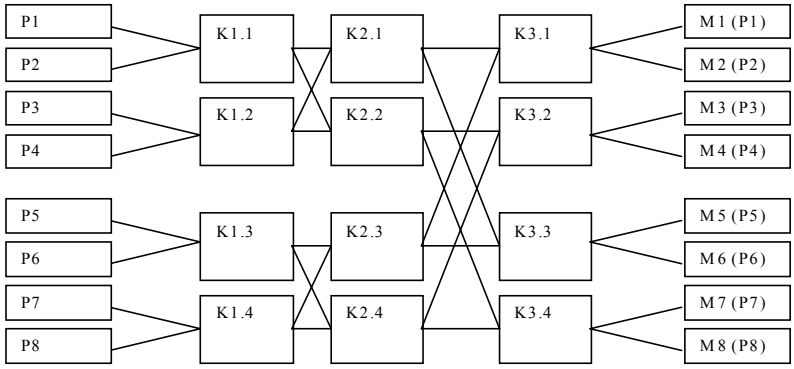

Рис. 6. Коммуникационная сеть типа «бабочка»

Ряд современных систем построен по кластерному принципу: процессоры и оперативная память разбиваются на несколько групп - кластеров. Внутри кластера процессоры имеют быстрый доступ к оперативной памяти. Доступ процессоров одного кластера к оперативной памяти, расположенной в другом кластере, так-же возможен, но время доступа при этом значительно возрастает.

Конфликты, неизбежно возникающие при записи разными процессорами одних и тех же данных, носят фундаментальный характер и снижают производительность системы за счет потерь на синхронизацию, независимо от конкретной аппаратной реализации системы.

Сказанное выше приводит к тому, что сконструированная система, как правило, не предусматривает возможности существенного наращивания числа процессорных узлов и обуславливает крайне высокую, относительно рассматриваемых в следующем разделе систем с раздельной памятью, стоимость.

Системы с раздельной памятью

Масштабируемые системы массового параллелизма конструируют на основе объединения каналами передачи данных процессорных узлов, обладающих своей локальной оперативной памятью, недоступной другим процессорам (рис. 7). Обмен данными между процессорами при таком подходе возможен лишь с помощью сообщений, передаваемых по каналам связи. Такая схема обладает рядом преимуществ по сравнению с системами, построенными на основе общей памяти. Подчеркнем основные преимущества систем с распределенной памятью.

- Сравнительно низкая стоимость - наименьший показатель отношения цена/производительность.
- Масштабируемость - возможность построения систем требуемой производительности, и наращивания их мощности за счет установки дополнительных процессоров.

Системы с раздельной памятью, по-видимому, всегда будут лидировать по показателю пиковой производительности, поскольку любые новые однопроцессорные (или многопроцессорные на основе общей памяти) системы могут быть легко объединены сетью и использованы в качестве многопроцессорных комплексов с раздельной памятью.

Но, к сожалению, эффективное использование систем с распределенной памятью требует значительных усилий со стороны разработчиков прикладного обеспечения и возможно далеко не для всех типов задач. Для широкого круга хорошо зарекомендовавших себя последовательных алгоритмов не удается построить эффективные параллельные аналоги.

Рис. 7. Многопроцессорная система с раздельной памятью

Рассмотрим ряд характерных представителей систем с раздельной памятью.

Транспьютеры

Наш обзор мы начнем с транспьютеров, с которых, собственно, и началось массовое распространение многопроцессорных систем. Типичная транспьютерная система выступает в качестве параллельного вычислительного ускорителя для какого-либо компьютера общего назначения - хост-компьютера (HOST). В качестве хост-системы с одинаковым успехом выступают как рабочие станции типа Sun, так и персональные компьютеры IBM PC (рис. 8).

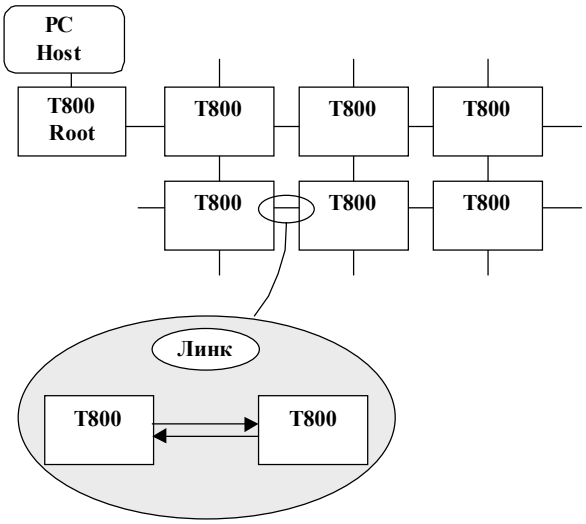

Рис. 8. Транспьютерная система и каналы связи

Каждый транспьютер производства фирмы INMOS (Великобритания) является полноценным процессором (рис. **9**), отличающимся от обычного процессора тем, что в своем составе он содержит 4 канала межпроцессорного обмена данными - линка (link). Каждый линк представляет собой устройство синхронного небуферизованного последовательного обмена данными по 4-проводной линии связи (сравните с 96-проводной линией доступа к оперативной памяти в системах с общей памятью). Именно благодаря возможности соединять между собой произвольное число процессоров, управляемых одной или несколькими хост-системами, транспьютерные системы получили столь широкое распространение.

В состав транспьютера типа T800 входит устройство обработки целых чисел (CPU), устройство обработки вещественных чисел и четыре двунаправленных канала передачи данных - линка. Каждое из перечисленных шести устройств может работать одновременно с другими и независимо от них. Выполняя запросы на обработку или передачу данных, каждое из них обращается к переменным, расположенным в доступной всем этим устройствам общей оперативной памяти. При этом вполне возможна ситуация, когда несколько устройств одновременно обращаются к одной и той же переменной. Запросы будут обработаны корректно, если данные только считываются, но если хотя бы одно из устройств выполняет запись данных, то результат выполнения операций чтения/записи становится неопределенным. Точно так же, не определен результат одновременной записи несколькими устройствами разных значений в одну и ту же переменную, в этом смысле каждый транспьютер является системой с общей памятью.

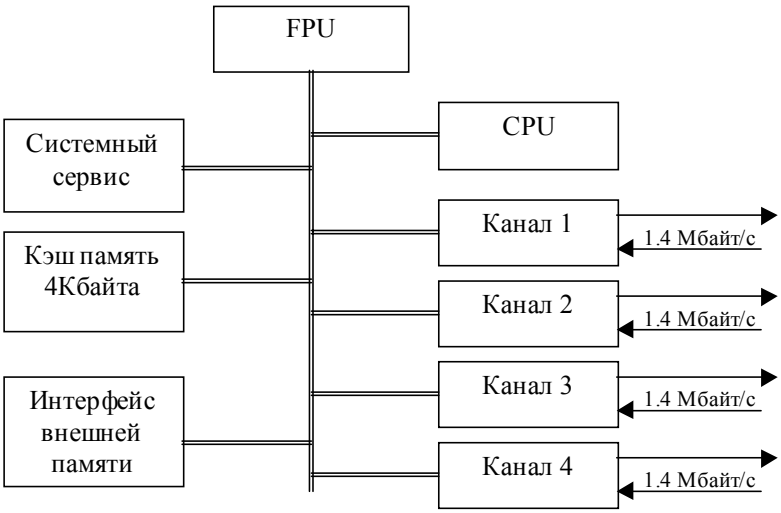

Рис. 9. Структура транспьютера типа T-800

В первых транспьютерных системах транспьютеры соединялись линками непосредственно между собой и получаемая таким образом конфигурация - топология, оказывалась зафиксирована на время решения задачи. При необходимости получения другой топологии линки приходилось соединять вручную. Для упрощения обслуживания и для обеспечения возможности изменения топологии системы непосредственно в процессе вычислений был разработан электронный коммутатор C-004 (рис. 10). Это электронно-конфигурируемый коммутатор, позволяющий задавать произвольные парные связи между 32 входами транспьютерных линков и 32 выходами. В приведенном на рис. 10 примере восемь транспьютеров жестко соединены в линейку (pipe) двумя из четырех своих линков, а остальными подключены к коммутатору. Между подключенными к коммутатору линками можно программно определить любые 4 связи. Например, показанная пунктиром связь превращает исходную топологию Линейка в топологию Кольцо.

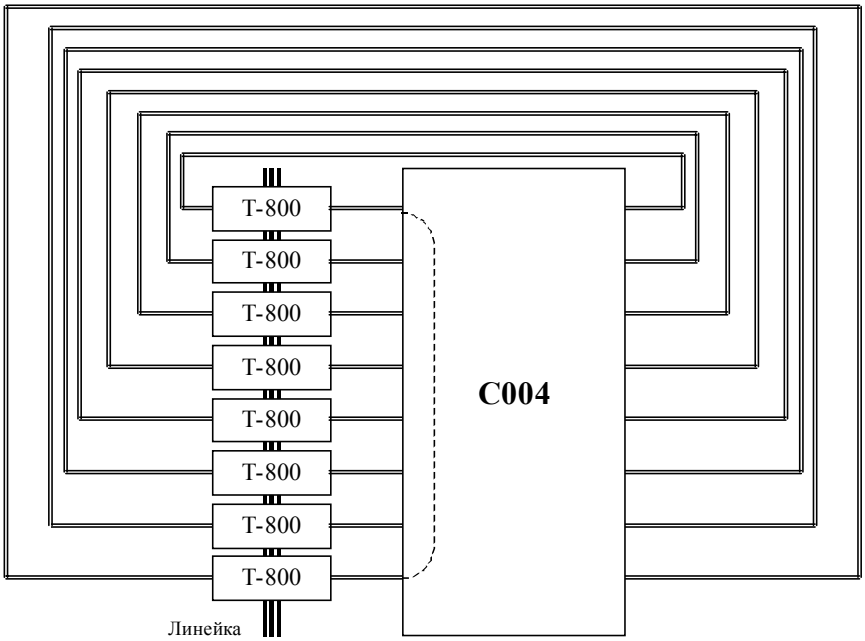

Рис. 10. Электронно-реконфигурируемое соединение транспьютеров с помощью коммутатора C004

Обладая сравнительно небольшой производительностью, транспьютеры быстро потеряли свое значение в качестве вычислительных элементов, но долгое время сохраняли его в качестве коммутационных в составе гибридных вычислительных систем.

Гибридные системы

Рассмотрим многопроцессорную систему Parsytec PowerXplorer, состоящую из 12 процессорных узлов (рис. 11). Каждый узел базируется на процессоре PowerPC-601. Реальная производительность каждого процессора - около 30 Mflops. Каждый процессор располагает 8 Mb локальной оперативной памяти. К этой же оперативной памяти имеет доступ транспьютер типа T805. Транспьютеры с помощью линков связаны между собой в прямоугольную сетку, часть транспьютеров имеет выход через один из своих каналов связи на управляющую машину типа SunSparc (рис. 12). Подключение осуществляется с помощью SCSI-устройства SCSI-Bridge, поддерживающего 4 транспьютерных линка для связи с системой PowerXplorer. Реальная производительность каждого из 4-х транспьютерных каналов связи - от 1 Мбайт/с при передаче длинных сообщений, до 0.3 Мбайт/с при передаче сообщений длиной в несколько байт.

Программирование возможно в рамках системы программирования PARIX™. Эта система позволяет использовать любую удобную виртуальую топологию (тор, решетку, звезду и т.д.), при этом транспьютер совершенно прозрачен для программиста. Физическая топология жестко задана и не изменяется.

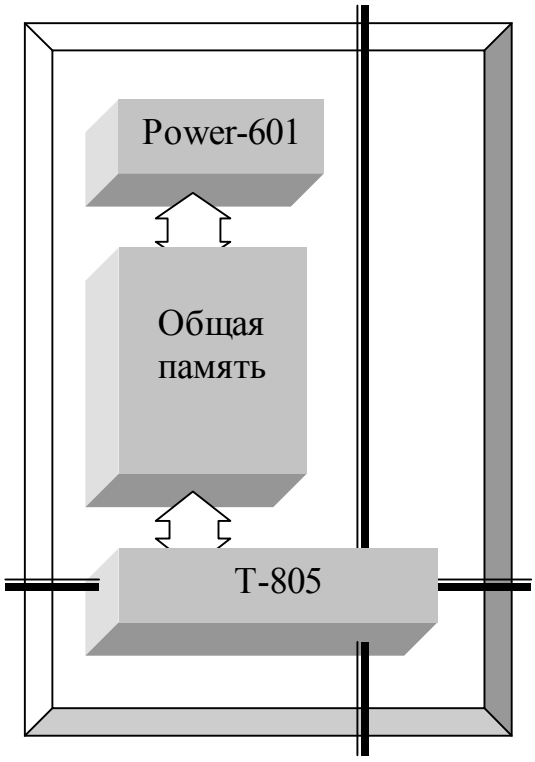

Рис. 11. Структура узла PowerXplorer

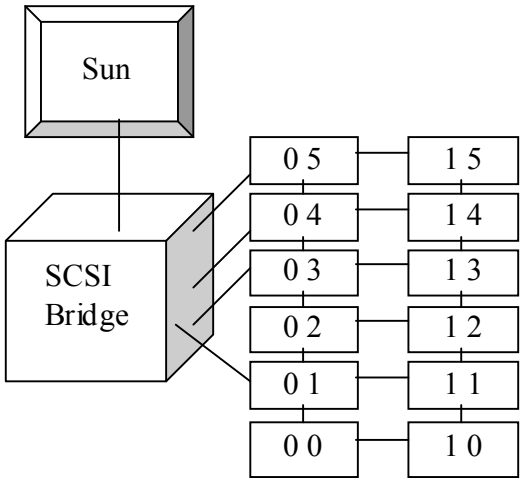

Рис. 12. Подключение 12-процессорной системы PowerXplorer к управляющей машине типа Sun

Гибридные системы, основанные на транспьютерных каналах связи, получили широкое распространение, но в настоящее время утрачивают свое значение из-за недостаточной пропускной способности линков.

Кластеры рабочих станций

С распространением локальных сетей получили свое развитие кластеры рабочих станций. Как правило, они представляют собой объединение небольшого числа персональных компьютеров и/или рабочих станций. Являясь сравнительно дешевым решением, эти системы часто проигрывают в эффективности обработки прикладных задач специализированным системам по следующим основным причинам:

- ряд распространенных локальных сетей (Ethernet, Token Ring) не поддерживают одновременную передачу данных между различными парами компьютеров в пределах одного сегмента сети. Это означает, что данные между компьютерами **c** и **d** могут быть переданы только после передачи данных между компьютерами **a** и **b**, что уменьшает и без того не очень высокую скорость передачи данных в таких сетях;
- практически всегда на рабочих станциях, составляющих кластер, продолжают выполняться последовательные задания пользователей. В результате менее загруженные процессоры вынуждены ожи-

дать более загруженные, что приводит к общему снижению производительности кластера при решении параллельной задачи до уровня, определяемого самой загруженной машиной. Эффективное решение этой задачи в условиях динамически изменяющейся, причем вне всякой зависимости от собственно параллельной программы, загруженности процессоров, представляется на сегодня весьма проблематичным;

- наличие в сети файловых серверов приводит к нерегулярно изменяющемуся объему данных, передаваемых через локальную сеть, что может значительно увеличивать время обмена сообщениями между процессорами, увеличивая интервалы простоя последних.

Системы на основе высокоскоростных сетей

Наиболее перспективными представляются многопроцессорные системы, построенные на основе специализированных высокоскоростных сетей передачи данных. Характерным представителем этого класса машин являются представленные на рис. 14, 15 системы типа Parsytec CC (Cognitive Computer - компьютер разумный). Каждый узел систем этой серии (рис. 13) представляет собой полноценный компьютер, управляемый UNIX-подобной операционной системой AIX. Система состоит из узлов двух типов - вычислительных узлов и узлов ввода/вывода. В состав процессорного узла входит вычислительный процессор Power PC-604, накопитель на жестком диске, адаптер локальной сети Ethernet, адаптер высокоскоростной сети HS_Link. В состав узла ввода/вывода дополнительно входят видеоадаптер, адаптеры клавиатуры и манипулятора типа мышь, жесткий диск увеличенного объема. Дополнительно может быть установлен второй адаптер сети Ethernet для подключения системы к локальной или глобальной сети. Показанный на рис. 13 последовательный интерфейс RS-232 служит для технического обслуживания системы. Каждый концентратор высокоскоростной сети (Router) может обслуживать до восьми HS-Link каналов.

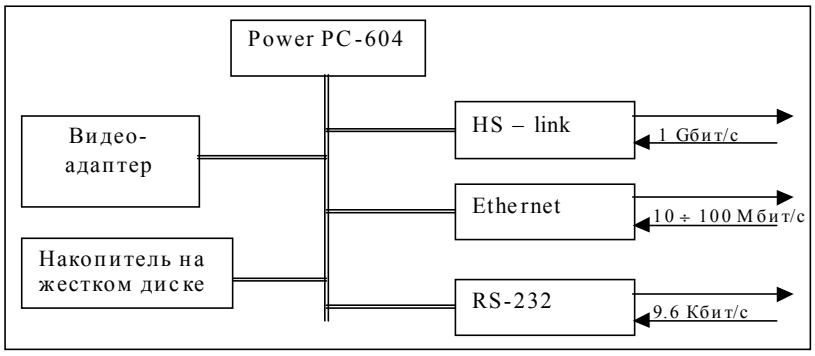

Рис. 13. Структура процессорного узла системы Parsytec CC

Таблица 1

Характеристики систем Parsytec CC-12,32

Характеристики\системы	Parsytec CC-12	Parsytec CC-32
Общее число процессоров	12	32
Кол-во проц. узлов	10	24
Кол-во двойных узлов ввода/вывода	1	4
Марка процессора	Power PC 604	Power PC 604
Тактовая частота процессора, MHz	133	100
Производительность процессора, Mflops	250	200
Оперативная память проц. узла, Мбайт	32	64
Оперативная память узла ввода-вывода, Мбайт	2x128	128+64
Дисковая память проц. узла, Мбайт	324	324
Дисковая память узла ввода-вывода, Gбайт	2x2	2x2
Внешняя дисковая память, Gбайт	2x4	2x9
Скорость межпроцессорных обменов, Мбайт/с	40	40
Суммарная производительность, GFlops	3	6.4
Суммарная оперативная память, Мбайт	576	2 304
Суммарная дисковая память для приложений, Gбайт	8	24
Суммарная пропускная способность, Мбайт/с	240	640
Тип подключения к локальной сети	Ethernet, 10 Мбит/с	Ethernet, 100 Мбит/с
Тип подключения к Internet	Ethernet, 10 Мбит/с	Ethernet, 10 Мбит/с

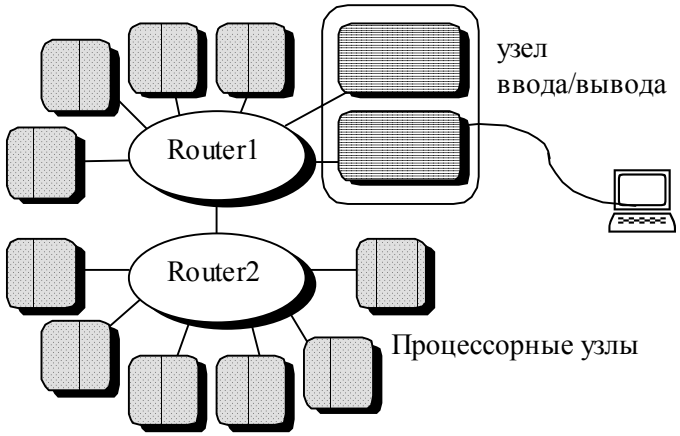

Рис. 14. Структура системы Parsytec CC-12

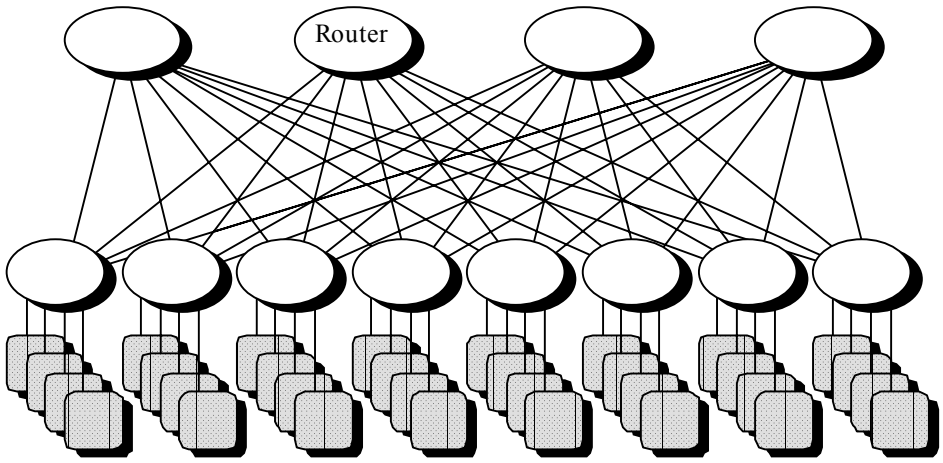

Рис. 15. Структура системы Parsytec CC-32

Основные характеристики систем Parsytec CC-12 и Parsytec CC-32 приведены в таблице 1. Схема соединения процессоров этих систем высокоскоростной сетью приведена на рис. 14, 15. Системы этой серии обладают следующими особенностями:

1) высокая скорость передачи данных между процессорами за счет применения специальных высокоскоростных коммуникационных модулей и высокоскоростных линков;

2) каждый вычислительный узел обслуживает собственная копия операционной системы, таким образом, каждый узел является независимой однопроцессорной вычислительной системой со своей дисковой памятью;

3) для функционирования системы не требуется наличия дополнительной управляющей машины, система сама по себе является полноправным членом локальной Ethernet сети.

4) связь между узлами осуществляется как по стандартной внутренней Ethernet сети, так и по специальной высокоскоростной сети HS-link.

Как уже отмечалось, системы на основе распределенной памяти являются масштабируемыми. Для объединения большого числа процессоров может использоваться легко масштабируемая, неограниченно наращиваемая топология, приведенная на рис. 16.

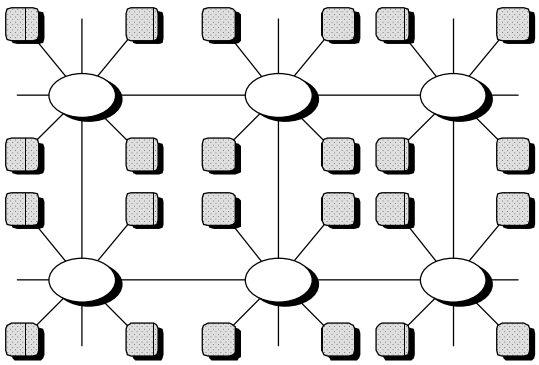

Рис. 16. Масштабируемый вариант топологии системы Parsytec CC

Еще одной особенностью систем этого типа, является возможность их построения без концентраторов высокоскоростной сети (Routers). Для этого достаточно установить на каждый из внутренних узлов цепочки процессоров, показанных на рис. 17, по два адаптора HS-link. С точки зрения программиста, все эти топологии идентичны, если они построены с использованием одинакового числа узлов, и представляются ему линейкой процессоров, пронумерованных последовательно, начиная с 0.

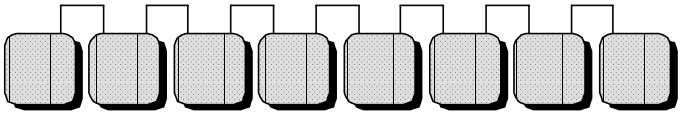

Рис. 17. Вариант топологии системы Parsytec CC, не требующий использования высокоскоростных коммутаторов.

Топологии многопроцессорных систем

Под топологией многопроцессорной системы будем понимать способ соединения процессорных узлов между собой каналами передачи данных. Удобно представить топологию системы в виде графа, вершины которого соответствуют процессорным узлам, а ребра – каналам связи, соответственно. Условно топологии можно разделить на фиксированные и реконфигурируемые, с одной стороны, и на регулярные и нерегулярные – с другой. Среди регулярных широко используются топологии следующих типов:

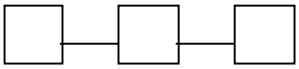

Рис. 18. Топология «линейка»

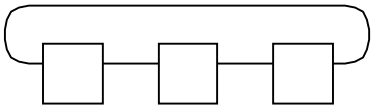

Рис. 19. Топология «кольцо»

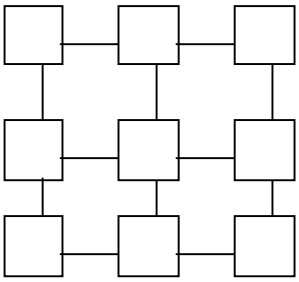

Рис. 20. Топология «решетка 3x3»

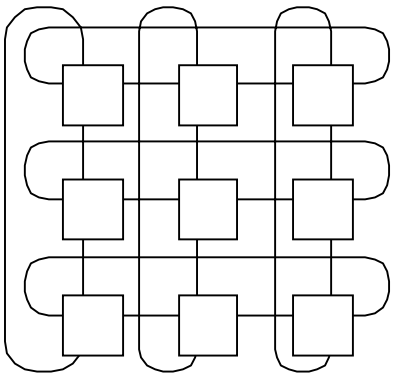

Рис. 21. Топология «тор 3x3»

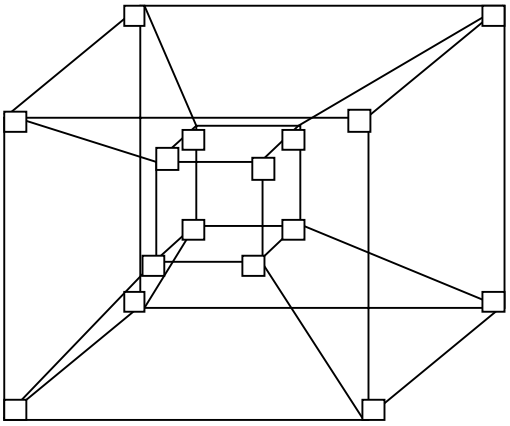

Рис. 22. Топология «гиперкуб степени 4»

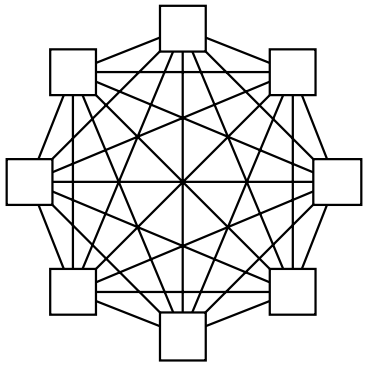

Рис. 23. Топология «клика»

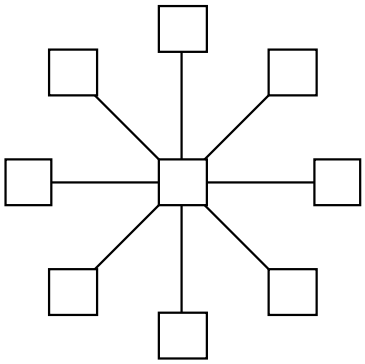

Рис. 24. Топология «звезда»

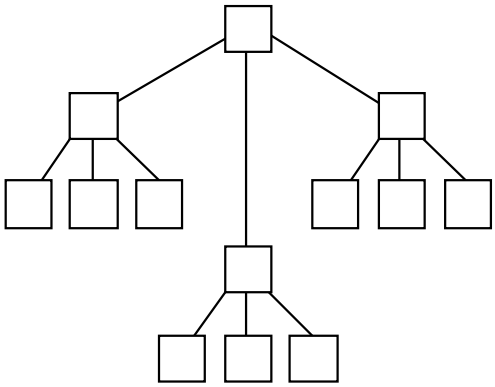

Рис. 25. Топология «троичное дерево»

Свойства используемой топологии определяют не только эффективность выполнения параллельной программы, но и возможность масштабирования самой вычислительной системы. Любое число процессоров может быть объединено в топологию типа Линейка, Кольцо, Клика. Однако для построения топологии типа Решетка или Тор требуется n1 x n2 процессоров, а значит, наращивание числа процессоров возможно только квантами размера n1 или n2. Для построения Гиперкуба требуется 2^n процессоров, это значит, во-первых, что каждая следующая система должна содержать вдвое больше процессоров, чем предыдущая, а во-вторых, требует для своей реализации наличия n каналов связей на каждом процессорном узле, что также ограничивает возможности наращивания числа узлов в системе. Для повышения эффективности выполнения программ на вычислительных системах необходимо согласовывать физическую топологию системы и топологию задачи. Значительная часть задач математической физики успешно решается на системах, процессоры которых объединены в решетки. Прямоугольные пространственные сетки, используемые для численного интегрирования систем дифференциальных уравнений, описывающих такие задачи, удобно делить на прямоугольные части, непосредственно отображаемые на решетку процессоров.

Существенно, что от физической топологии может сильно зависеть эффективность выполнения конкретной программы на ВС.

Первые многопроцессорные системы, получившие широкое распространение, обладали ограниченными возможностями в плане реконфигурирования. Системы на основе транспьютеров позволяли строить двумерные решетки процессоров, кольца, цилиндры и торы. Однако торами то, что можно было построить на основе транспьютеров, можно назвать условно: для реализации тора нужно именно 4 линка, при этом не остается линка для подсоединения системы к управляющей вычислительной машине (Рис. 26).

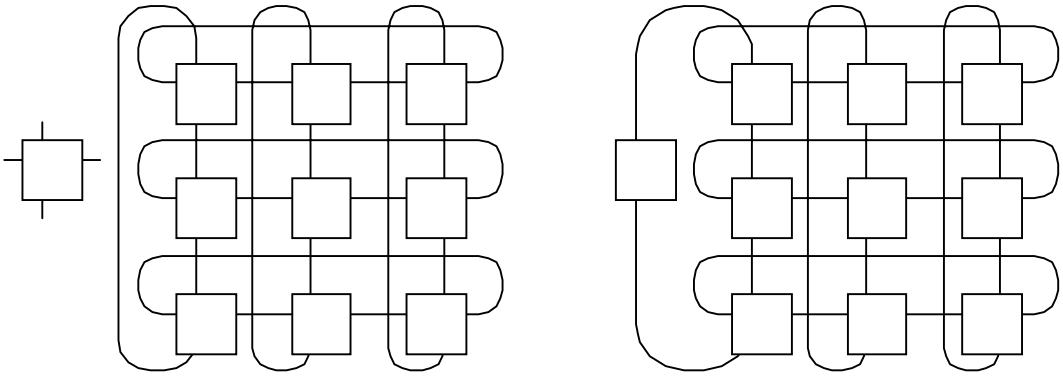

Рис. 26. Подключение тора процессоров к управляющей машине

В результате нарушается однородность процессов передачи данных между процессорами во время вычислений. Построить на таких элементах гиперкуб размером более $2^4 = 16$ процессоров, или трехмерный куб нельзя, не объединяя процессоры в блоки. Например, объединяя процессоры попарно на основе общей памяти в блоки по два процессора, получим вычислительные узлы с 8-ю линками (рис. 27).

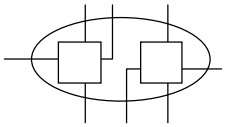

Рис. 27. Объединение процессоров в вычислительные узлы

Обращает на себя внимание тот факт, что и гиперкуб и трехмерный куб содержат в своем составе решетку. Возникает актуальный вопрос: какой должна быть физическая топология для того, чтобы при наличии весьма жестких ограничений на число каналов получить минимальное расстояние между наиболее удаленными процессорами? Иными словами: как можно минимизировать диаметр графа процессоров, сохраняя ряд связей, необходимых для эффективного выполнения программы? Возможное частное решение – построение топологий типа «пирамида» (рис. 28, 29). Они дают наилучший из возможных результатов, но не являются масштабируемыми.

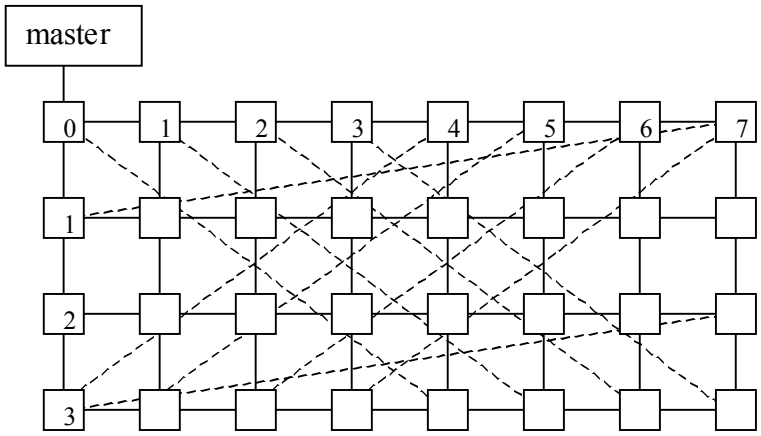

Рис. 28. Пример графа из 32 процессоров с диаметром и радиусом равными 4

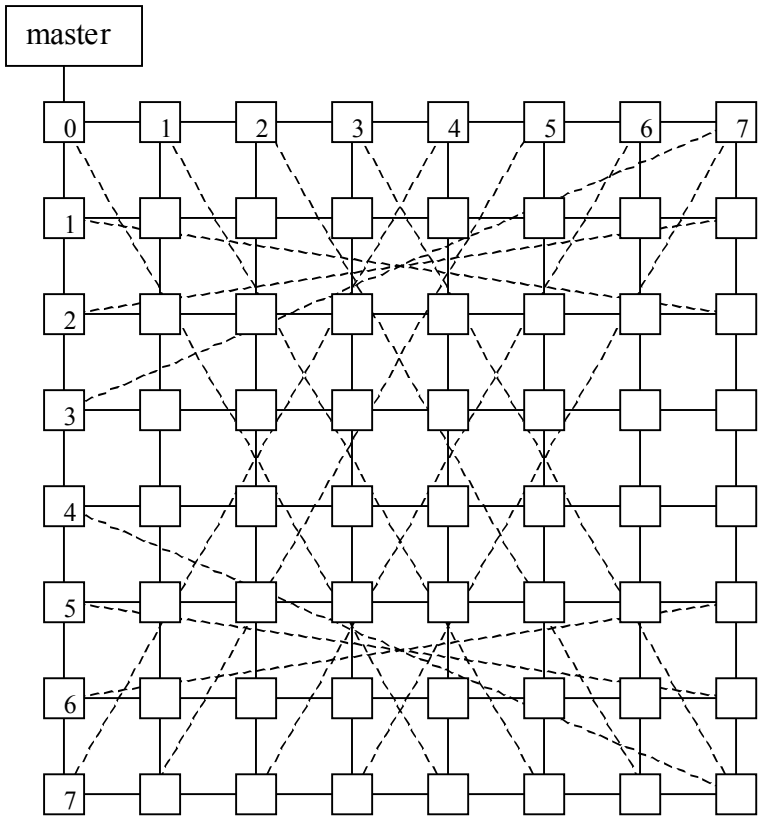

Рис. 29. Пример графа “пирамида” из 64 процессоров с диаметром и радиусом равными 6

Тема 2. Принципы построения параллельных алгоритмов

Рассматривая параллельное программирование, будем следовать подходу, предложенному Хоаром (*C A R Hoare.* Communicating Sequential Processes) и развитому Э.Дейкстрой (*Dijkstra E.W.* Cooperating Sequential Processes) в работе «Взаимодействие последовательных процессов». В соответствии с ним, параллельный алгоритм можно представить в виде ансамбля последовательных взаимодействующих процессов.

Построение параллельной программы на сегодня ближе к искусству, чем к формализованному процессу. Практически неизвестны априорные способы установления правильности параллельного алгоритма, не говоря об установлении корректности конкретной реализации алгоритма в виде программы на том или ином алгоритмическом языке. И если верно утверждение, что в любой последовательной программе есть, по крайней мере, одна ошибка, то для параллельной программы это утверждение еще более справедливо. Если для последовательной программы сохраняется призрачная надежда построения системы тестов, выполнение которых в некоторой мере гарантирует работоспособность программы на некотором подмножестве исходных данных, то для параллельной программы мы не имеем даже такой воз-

можности. Одна и та же программа, на одних и тех же данных может неограниченно долго давать верные (ожидаемые от нее) результаты, но дать совершенно непредсказуемый результат при очередном запуске с теми же данными. Причина такого неадекватного поведения параллельных программ кроется в наличии неопределенности в порядке выполнения действий, необходимых для получения результата. Например, ожидая ввода двух величин с двух каналов, мы не можем заранее предсказать, с какого канала данные придут раньше. Соответственно неизвестно, чему будут равны значения переменных, зависящих от этого порядка, по получении данных. Не имея сегодня возможности доказывать правильность параллельных алгоритмов и программ в общем случае, мы, тем не менее, можем конструировать вполне работоспособные алгоритмы, основываясь на некоторых изложенных ниже подходах к их построению. Ограничивая собственную свободу в выборе структуры алгоритма, мы можем взамен, опираясь на здравый смысл и опыт использования излагаемых методов, быть уверенными в корректности получаемых параллельных программ.

Виды параллелизма

Укажем основные типы параллелизма, используемые для построения параллельных алгоритмов:

1. алгоритмический;
2. геометрический;
3. конвейерный;
4. «коллективное решение», он же «процессорная ферма».

Излагаемые принципы являются надежной основой для успешного решения широкого круга задач и дальнейшей разработки методов построения параллельных алгоритмов.

Алгоритмический параллелизм

Довольно часто алгоритм распадается на несколько достаточно независимых крупных частей, которые можно выполнять одновременно. Например, задача управления роботом, позволяет выделить отдельные микропроцессоры для управления его различными частями, такими как манипуляторы, устройства ввода информации, система передвижения. При этом между процессорами требуется минимальная синхронизация, выполнять которую может отдельная программа. Важно, что в этом случае каждый процессор выполняет свою, возможно уникальную программу, и число используемых процессоров определяется, таким образом, числом независимых частей. Увеличить число задействованных процессоров с целью увеличения скорости принятия решения при таком подходе затруднительно. Фактически система такого рода не является масштабируемой. Другое дело, что увеличение общего числа процессоров путем введения дублирующих процес-

соров позволяет строить отказоустойчивые системы, что представляет самостоятельный интерес и находит широкое применение, например, при реализации критических технологий на производстве, в аэрокосмической технике, телекоммуникационных системах. Для нас наибольший интерес будут представлять масштабируемые алгоритмы, к рассмотрению основных принципов построения которых мы и переходим.

Стена Фокса

Рассмотрим подробнее основные способы создания масштабируемых параллельных программ на примере известной задачи построения «стены Фокса» - длинной стены, состоящей из прямоугольных кирпичей, расположенных в соответствии с рис. 30.

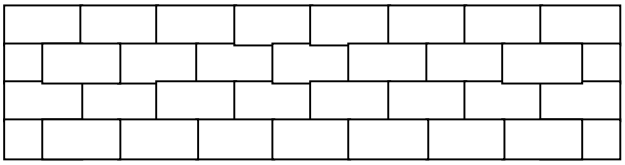

Рис. 30. Стена Фокса

Геометрический параллелизм

Геометрический параллелизм является одним из наиболее распространенных типов параллелизма, применяемых при решении задач математической физики, обработки изображений и других задач, где имеется большое число однородных действий над однородными данными, которые можно так разбить на группы, что при обработке каждой группы не потребуется обращений к данным из других групп, за исключением их малого числа.

Последнее свойство будем называть свойством локальности алгоритма. Например, алгоритм решения системы линейных уравнений свойством локальности не обладает, косвенным подтверждением чего может служить тот факт, что изменение любого из коэффициентов может привести к кардинальному изменению всего решения. Можно сказать, что все переменные сильно зависят друг от друга. Напротив, алгоритм моделирования системы материальных точек, соединенных в цепочку упругими пружинками, свойством локальности обладать будет, поскольку при длинной цепочке изменение положения одной из точек или упругости одной из пружинок на левом конце цепочки не сразу скажется на поведении ее правого конца.

Локальный алгоритм допускает разбиение данных на части по числу процессоров, при котором обработку каждой части осуществляет соответствующий процессор. Этот подход весьма эффективен при условии, что действия, выполняемые одним процессором, зависят лишь от небольшого, ограниченного объема данных, расположенных на других процессорах. Желательно, что бы эти «чужие» данные были

локализованы на ограниченном, небольшом количестве процессоров. Например, разбиение области рисунка прямоугольной сеткой позволит в первом приближении, при анализе изображения на решетке процессоров, ограничиться обменом данными только с ближайшими в решетке четырьмя процессорами.

Решая задачу построения «стены Фокса», можно разбить стену на равные по длине участки и поручить постройку каждого участка отдельному каменщику. В этом случае все каменщики могут начать работу одновременно, укладывая нижний слой кирпичей (рис. 31). Перед укладыванием очередного слоя каждому каменщику следует убедиться, что кирпичи предыдущего слоя уложены не только на его участке, но и на прилегающих участках. Если работа на соседних участках еще не закончена, возникают вынужденные паузы, связанные с синхронизацией работ на соседних участках. Отметим, что паузы могут возникать, даже если каменщики работают с одинаковой скоростью, поскольку объем работ, вообще говоря, зависит от номера участка, например он разный для крайних и внутренних частей стены.

Можно сделать вывод о том, что эффективность организации параллельной работы будет тем выше, чем длиннее участки стены, и при достаточно длинных участках следует ожидать эффективности, близкой, но все же меньшей 1. При коротких участках стены (очень много каменщиков) эффективность будет невысокой, поскольку время независимой работы каждого каменщика будет невелико, они будут много времени проводить в ожидании друг друга на стыках участков.

Отметим, что этот метод организации работы эффективен, даже если стена не очень высокая. Главным фактором, определяющим эффективность, является ее достаточная протяженность, по отношению к числу задействованных каменщиков.

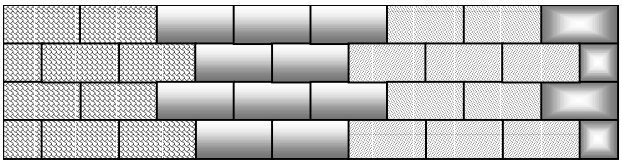

Рис. 31. Геометрическое решение

Конвейерный параллелизм

Конвейерное распределение работ предполагает, что первый каменщик полностью укладывает первый слой кирпичей, второй каменщик – второй слой и так далее. После того, как первый полностью уложит свой первый слой, он продолжает укладку очередного ***p+1***-го слоя (***p*** – общее число каменщиков). Очевидно, что при таком подходе сразу может начать работу только первый каменщик, остальные вынуждены ожидать, пока будут уложены нижние слои. Дольше всех будет ожидать своей очереди последний каменщик. Аналогичная ситуация сложится в конце работы - первый каменщик закончит работу первым и будет простаивать, пока не закончат остальные. Если стена

достаточно высокая и длинная, большую часть времени все каменщики будут равномерно загружены и простоев, связанных с ожиданием друг друга, быть не должно.

Последнее утверждение верно при условии, что все каменщики работают с одинаковой скоростью, и что объем работ не меняется от слоя к слою стены. На практике всегда есть некоторая разница, как в скорости работы каменщиков, так и во времени, необходимом для обработки каждого слоя. Очевидно, что при достаточно высокой стене, рано или поздно скорость ее построения будет определяться самым медленным каменщиком, поскольку остальные не смогут продолжать работу, пока не уложены предыдущие слои.

Основной вывод выглядит следующим образом: метод эффективен при большом, по сравнению с числом задействованных каменщиков, количестве однородных фрагментов работ и при равной производительности каменщиков.

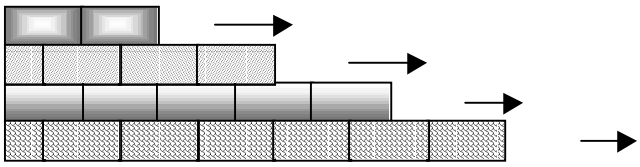

Рис. 32. Конвейерное решение

Параллелизм типа «коллективное решение»

Возникает закономерный вопрос, как поступить в случае, когда априори нельзя сказать сколько-нибудь определенно о соотношении производительности каменщиков, и об относительной трудоемкости разных участков работы. Метод «коллективного решения» позволяет эффективно организовать работу именно в этом случае. Он не предполагает априорного жесткого распределения работ между работниками. Каждый получает некоторое количество кирпичей и цемента и укладывает любой готовый к укладке незанятый участок стены. Израсходовав свой запас материалов, он, получив следующую порцию, приступает к укладке некоторого, в общем случае, другого, участка стены. Легко указать основные моменты, приводящие к непроизводительному простою каменщиков при таком подходе. Во-первых, требуется синхронизация при получении материалов и при выборе участков стены – каждому должен предоставляться подготовленный участок достаточной длины, а это значит, что неизбежны некоторые простои в начале работы при распределении участков. Точно так же вероятен некоторый простой в конце работы, когда кто-то закончит раньше остальных.

При достаточно длинной стене и достаточно быстром механизме распределения работ подход позволяет поддерживать высокую эффективность. Однако при большом числе каменщиков, именно механизм распределения работ может, и часто оказывается, основным фактором, ограничивающим общую эффективность. Вспомним, что в контексте реальной вычислительной системы, передача части работ от управляющего к обрабатывающему процессору связана с передачей определенного объема данных по каналам межпроцессорной связи.

Это, в свою очередь, означает, что число обрабатывающих процессоров не может превышать отношения времени обработки одной порции данных ко времени приема/передачи данных, связанных с этой порцией.

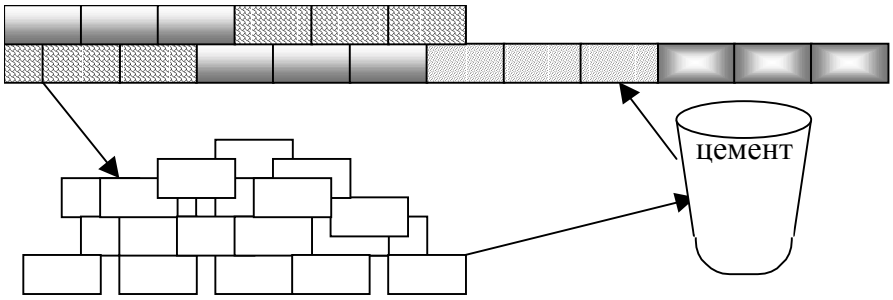

Рис. 33. Коллективное решение

Эффективность и ускорение параллельных алгоритмов

В связи с высокой трудоемкостью проектирования и отладки программ для систем массового параллелизма актуально иметь возможность предварительной оценки степени пригодности алгоритмов для их выполнения на многопроцессорных системах. Соответствующую количественную оценку дают понятия эффективности и ускорения параллельных алгоритмов. Как уже говорилось, на системе из ***p*** процессоров ту или иную задачу нельзя решить в ***p*** раз быстрее, чем на однопроцессорной системе. Связано это с тем, что, во-первых, практически никакой алгоритм не позволяет на всех своих этапах эффективно использовать все процессоры системы, во-вторых, всегда есть затраты времени на обмен данными между процессорами, увеличивающие время выполнения параллельного алгоритма, и, в-третьих, параллельный алгоритм всегда содержит некоторое количество дополнительных действий, связанных с управлением параллельной программой и синхронизацией ее частей, что также увеличивает общее время вычислений.

Основной характеристикой алгоритма, определяющей эффективность его выполнения на многопроцессорной системе является его степень параллелизма.

Степенью параллелизма алгоритма называют число действий алгоритма, которые могут выполняться одновременно, каждое на своем процессоре.

Рассмотрим задачу сложения двух векторов **a** и **b,** длины n.

Каждое из сложений

$c_i = a_i + b_i$, $i=1, ..., n$

не зависит от других и может выполняться одновременно с ними, таким образом, степень параллелизма данного алгоритма равна n. Подчеркнем, что степень параллелизма не зависит от числа процессоров, на котором предполагается выполнять соответствующую программу. Степень параллелизма является внутренним свойством выбранного алгоритма. От числа процессоров зависит только время выполнения реальной программы при ее запуске на этих процессорах. Можно сделать вывод о том, что ускорить выполнение программы бо-

лее чем в число раз, определяемое степенью параллелизма ее алгоритма, нельзя.

Ускорением параллельного алгоритма (S_p), называют отношение времени выполнения алгоритма на одном процессоре ко времени выполнения алгоритма на системе из ***p*** процессоров.

С ускорением параллельного алгоритма тесно связано понятие его эффективности.

Эффективностью параллельного алгоритма (E_p) называют отношение его ускорения к числу процессоров, на котором это ускорение получено:

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

Рассмотрим в качестве примера задачу определения суммы элементов массива:

Определить сумму элементов массива a, содержащего n членов.

Последовательный алгоритм решения этой задачи требует одной операции инициализации и *n-1* операции сложения:

$s=a_1$

$s=s+a_i, i=2, ..., n.$

Построим параллельный алгоритм, используя метод сдваивания, в соответствии со схемой рис. 34.

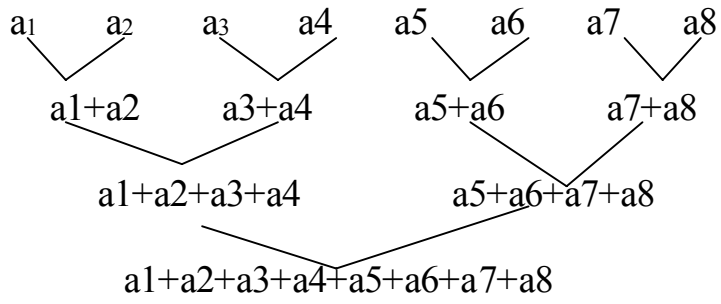

Рис. 34. Определение суммы элементов массива чисел на четырех процессорах методом сдваивания

Для массива размером $n=2^q$ алгоритм сдваивания содержит $q=\log_2 n$ этапов. На первом этапе выполняется параллельно $n/2$ действий, на втором - $n/4$, …, на последнем – одно. Таким образом, степень параллелизма меняется от этапа к этапу и на этапе k равна $n/2^k$. Определим ускорение алгоритма сдваивания на системе из $p=n/2$ процессоров:

$$S_p = \frac{nt}{(2+2\alpha)t\log_2 n + 2t} = \frac{p}{(1+\alpha)\log_2 p + 1},$$

$$E_p = \frac{1}{(1+\alpha)\log_2 p + 1},$$

где t - время выполнения одного сложения, αt – время переноса одного числа от одного процессора к другому.

При определении ускорения учтены следующие соображения:

- время решения задачи на одном процессоре складывается из: одной операции инициализации и (n-1) операций сложения;
- время решения задачи на $p=n/2$ процессорах складывается из: одной операции инициализации и одной операции сложения на каждом этапе;
- всего этапов $\log_2 n$;

при параллельном решении на каждом этапе процессоры у которых есть данные выполняют одну операцию инициализации и одну операцию сложения, всего – 2 операции на этап. Таким образом, тот процессор, который выполнит последнее сложение, совершит всего $2\log_2 n$ операции инициализации и сложения, затратив на это время $2t\log_2 n$.

После этапа k следует передать на каждый из процессоров, участвующий в решении на этапе k+1, два числа, затратив время $2\alpha t$. Передавать данные от этапа к этапу надо $\log_2 n$-1 раз, значит общее время, затраченное на передачу данных: $2\alpha t(\log_2 n-1)$, общее время сложения массива чисел длины n на системе из p процессоров составляет, таким образом:

$$2t\log_2 n + 2\alpha t(\log_2 n-1) = 2t(\log_2 p+1)+2\alpha t\log_2 p.$$

Рис. 35. Ускорение параллельного алгоритма сложения массива чисел методом сдваивания

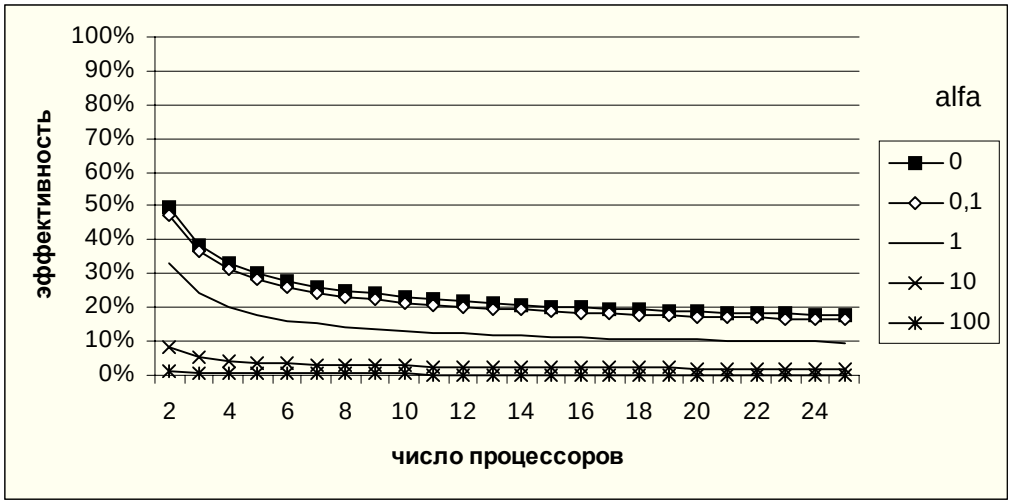

Рис. 36. Эффективность параллельного алгоритма сложения массива чисел методом сдваивания

Графики, соответствующие различным значениям параметра α, приведены на рис. 35, 36. Напомним, что этот результат получен в предположении, что объем вычислительной работы растет как степень числа процессоров. Можно сделать вывод, что даже при отсутствии расходов времени на передачу данных между процессорами (график $\alpha=0$), эффективность алгоритма не превышает уровня 50%, использование же этого алгоритма на современной технике, для которой характерны $\alpha>1$, фактически сводит на нет преимущества параллельного решения задачи с помощью рассмотренного алгоритма.

Рассмотрим более эффективное решение этой же задачи с помощью метода геометрического параллелизма (рис. 37).

Будем действовать следующим образом:

- разобьем массив длины n на p равных частей и выполним сложение n/p чисел на каждом процессоре
- передадим результаты на один из процессоров и выполним там сложение p полученных чисел.

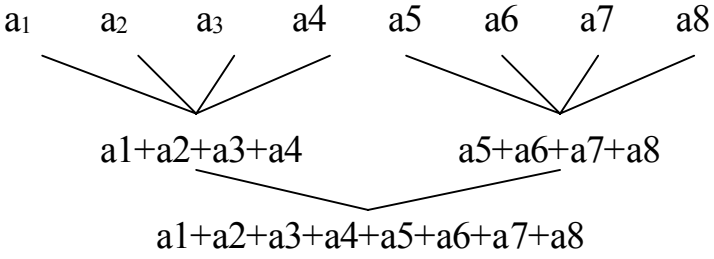

Рис. 37. Определение суммы элементов массива чисел на двух процессорах методом геометрического параллелизма

Ускорение и эффективность, достигаемые применением данного алгоритма, будут иметь вид:

$$S_p = \frac{nt}{\frac{n}{p}t + (1+\alpha)tp} = \frac{np}{1+(1+\alpha)p^2},$$

$$E_p = \frac{S_p}{p} = \frac{n}{1+(1+\alpha)p^2},$$

соответствующие графики, в предположении фиксированного числа элементов в массиве (n=4096), приведены на рис. 38, 39.

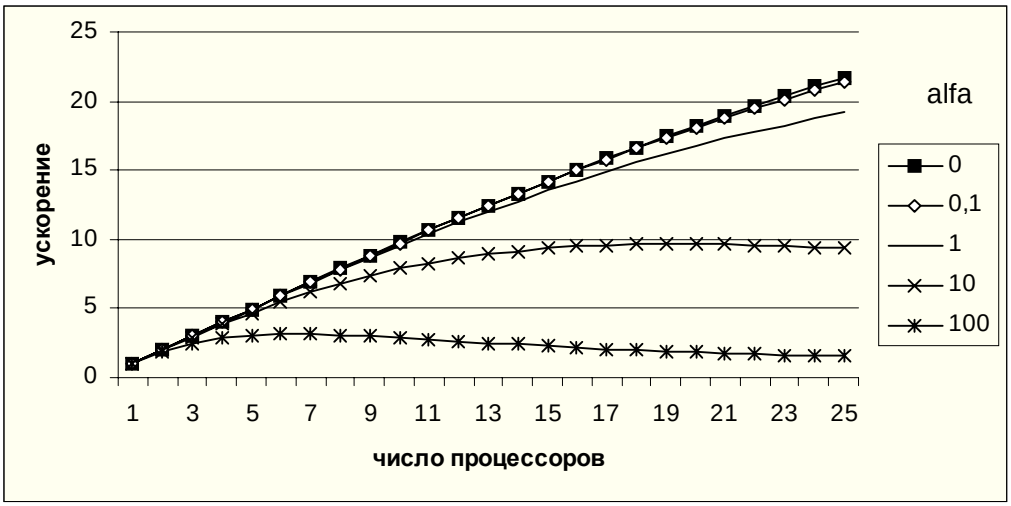

Рис. 38. Ускорение параллельного алгоритма определения суммы элементов массива чисел методом геометрического параллелизма

Из рис. 39 следует вывод о том, что при увеличении числа процессоров, используемых для решения задачи фиксированного размера (в данном случае фиксированного числа элементов в массиве), эффективность, начиная с некоторого момента, уменьшается. В общем случае существует некоторый оптимум числа процессоров, при котором система используется наиболее эффективно.

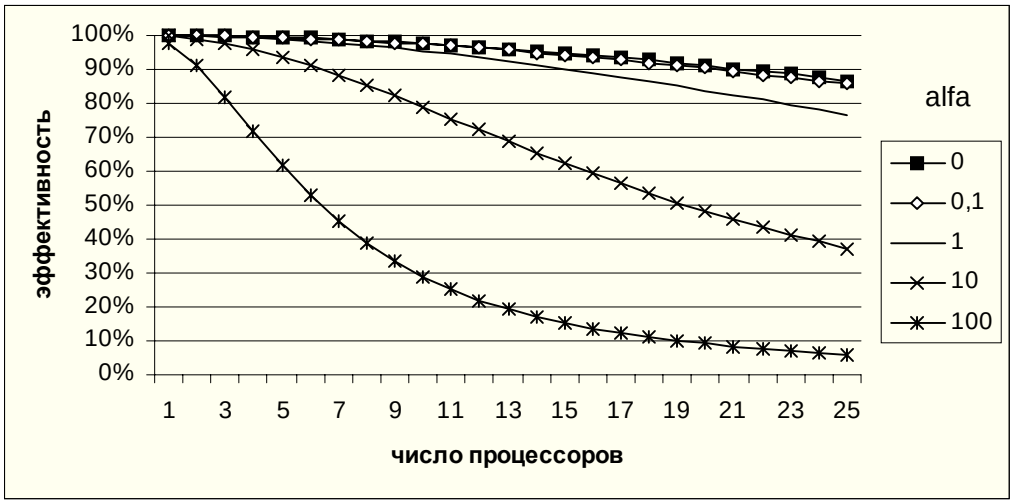

Рис. 39. Эффективность параллельного алгоритма определения суммы элементов массива чисел методом геометрического параллелизма

В приведенных рассуждениях предполагалось, что элементы массива изначально уже распределены по процессорам необходимым образом. Для достаточно широкого круга задач сделанное предполо-

жение действительно справедливо. Например, при моделировании с помощью явных разностных схем газодинамических течений выполняется достаточно много шагов, при которых каждый процессор использует данные, преимущественно расположенные непосредственно на нем или на соседних с ним процессорах, что не требует передачи какой-либо информации на управляющий процессор. Так же, при обработке изображений, в случае, когда каждый из процессоров получает информацию непосредственно со «своей» видеокамеры, предварительная рассылка данных не требуется.

В связи с полученным при увеличении числа процессоров падением эффективности уместно сделать одно принципиальное замечание. Вводя понятие эффективности мы положили размер задачи неизменным, что приводило к уменьшению, с ростом числа процессоров, вычислительной нагрузки на каждый процессор. Однако для широкого круга задач ситуация выглядит несколько иначе. Использование многопроцессорных систем позволяет проводить моделирование на подробных сетках, содержащих тем больше узлов, чем больше доступно процессоров. Можно говорить о том, что сохраняется не общее число узлов, т.е. размер задачи, а число узлов, приходящееся на один процессор, что приводит к сохранению высокой эффективности при увеличении числа используемых процессоров.

Сравнение с наилучшим последовательным алгоритмом

Не всегда параллельный алгоритм является наилучшим выбором для последовательной системы . Например, для многих прямых методов решения систем алгебраических уравнений не найдены эффективные параллельные аналоги. Вместе с тем, для этих целей успешно используются легко распараллеливаемые итерационные алгоритмы, уступающие прямым методам, если речь идет о решении на одном процессоре, но выигрывающие у последних за счет большей степени внутреннего параллелизма на многопроцессорных системах. В связи с этим актуально еще одно определение ускорения.

Ускорением параллельного алгоритма по сравнению с наилучшим последовательным называется отношение времени выполнения быстрейшего последовательного алгоритма на одном процессоре ко времени выполнения параллельного алгоритма на системе из p процессоров.

Эффективностью параллельного алгоритма по отношению к наилучшему последовательному называют отношение его ускорения по сравнению с наилучшем последовательным алгоритмом к числу процессоров, на котором это ускорение получено.

Закон Амдаля

Принимая во внимание, что параллельный алгоритм может на разных этапах обладать разной степенью внутреннего параллелизма, рассмотрим полезную модель поведения ускорения в зависимости от свойств алгоритма.

Обозначим через α долю операций алгоритма, выполнение которых невозможно одновременно с другими операциями – иными словами, долю операций, выполняемых со степенью параллелизма 1. Предполагая, что все операции в алгоритме выполняются либо последовательно , либо с максимальной степенью параллелизма, равной p, получим, что доля последних равна 1-α.

Запишем в общем виде ускорение, достигаемое на p процессорах:

$$S_p = \frac{T_1}{\left(\alpha + \frac{1-\alpha}{p}\right)T_1 + t_d},$$

где T_1 – время выполнения алгоритма на одном процессоре, t_d – общее время подготовки данных. Под временем подготовки данных будем понимать все дополнительные расходы на синхронизацию частей программы, передачу данных между процессорами, т.е. все операции, отсутствующие в последовательном алгоритме и не учтенные вследствие этого, множителем, стоящим в знаменателе при T_1.

Рассмотрим некоторые характерные случаи:

1. α=0, t_d=0, $S_p = p$.

Все операции выполняются с максимальной степенью параллелизма, ускорение равно p.

2. $\alpha\neq0$, t_d=0, $S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$.

Последнее соотношение выражает закон Амдаля или Уэра (Gene Amdahl, Ware). Он предполагает, что все действия выполняются либо с минимальным, либо с максимальным параллелизмом, причем задержки, связанные с подготовкой данных, отсутствуют.

Очевидный вывод состоит в том, что ускорение при сделанных предположениях меньше p (поскольку $S_p = \frac{p}{\alpha(p-1)+1} < p$).

Пусть $\alpha = \frac{1}{2}$, тогда $S_p = \frac{2}{1+\frac{1}{p}} < 2$. Независимо от числа процессоров ускорение меньше двух, если половина действий алгоритма должна выполнятся последовательно.

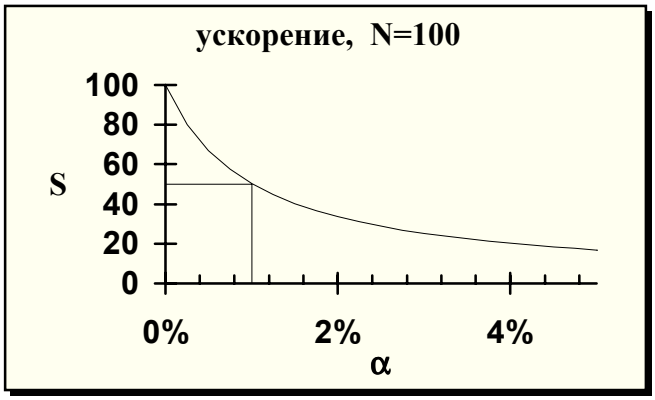

Рис. 40. Закон Амдаля

Рассмотрим поведение ускорения в зависимости от значения α при использовании 100 процессорной вычислительной системы (p=100). Соответствующий график приведен на рис. 40. График показывает быстрый спад ускорения при малых значениях доли операций, выполняемых последовательно . Помеченная на графике точка соответствует α=0.01, ускорение при этом оказывается равным 50. Всего 1% операций, не подлежащих распараллеливанию, вдвое снижает эффективность выполнения алгоритма на 100 процессорной системе.

3. α - любое, t_d>>0.

При большом времени подготовки данных независимо от значения α, определяющего возможности распараллеливания алгоритма, может оказаться, что $S \approx \frac{T}{t_d} < 1$. Большие затраты на обмен данными и синхронизацию частей программы могут привести к тому, что вместо многопроцессорной системы эффективнее окажется использование одного единственного процессора.

Пропускная способность каналов связи

Рассмотрим основные свойства каналов межпроцессорной связи на примере транспьютерных линков.

Эффективная скорость передачи данных через линк сильно зависит от объема сообщения и меняется в широких пределах - от 180 Кбайт/с до 1.7 Мбайт/с. Она минимальна при пересылке данных небольшой длины - отдельных переменных (1÷8 байт) или коротких массивов. Время, необходимое для передачи сообщения, можно определить, пользуясь данными, приведенными в табл. 2, где n - размер сообщения в байтах.

Из таблицы 2 следует, что максимально возможная скорость передачи достигается при обмене большими массивами. При уменьшении длины пересылаемых данных до 8 байт эта скорость падает более чем в два раза. Можно сделать важный практический вывод - следует сокращать не только общий объем передаваемых данных, но и

число пересылок “коротких” данных (короче 32 байт), для чего объединять их в крупные блоки.

Таблица 2

Передача данных через канал межпроцессорной связи

Время, мкс	Производительность	Вид операции
1 ÷ 2.3	1 ÷ 0.43 Mflops	Одна операция (+ - * /) с плавающей точкой.
10	385 Кбайт/с	Передача одного длинного целого числа через порт (4 байта).
12	629 Кбайт/с	Передача одного вещественного числа двойной точности через порт (8 байт).
0.57n+8	1.7 - 13/(.57n+8) Мбайт/с	Передача в одну сторону через порт.
0.076n+0.85	12.5 - 10.6/(0.076n + 0.85) Мбайт/с	Копирование данных между буферами внутри одной задачи.

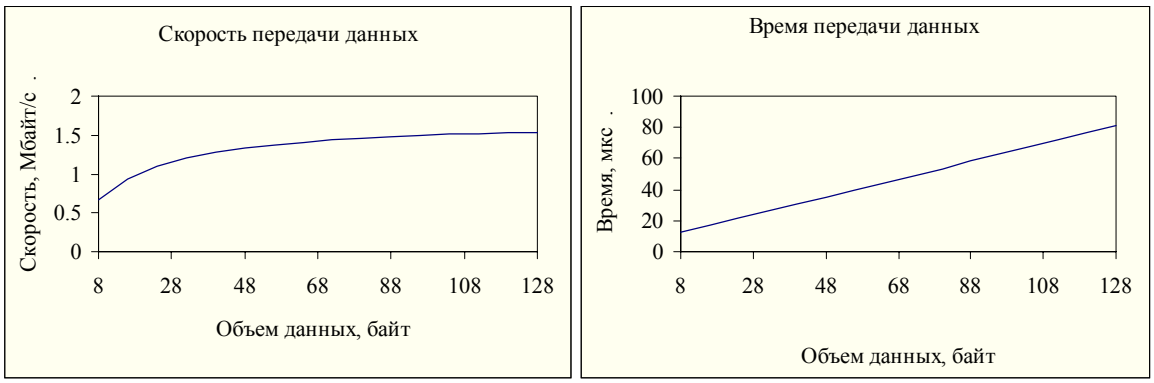

Рис. 41. Характеристики транспьютерного канала связи

Тема 3. Синхронизация последовательных процессов

Параллельное решение некоторой задачи на нескольких процессорах предполагает наличие между последними определенного взаимодействия. Прежде чем переходить к описанию средств, необходимых для обеспечения такого взаимодействия, рассмотрим известную задачу Дейкстры (*Dijkstra E.W.*) «Обедающие философы». На ее примере можно прокомментировать две фундаментальные проблемы – возникновение тупиков и недетерминированность параллельных вычислений.

Предупреждение: *Используемые в этой теме функции, типы переменных и некоторые синтаксические конструкции служат чисто иллюстративным целям и не всегда соответствуют стандартам языка С. Используемые функции, даже совпадающие по названию с существующими в библиотеке С или PARIX™, не соответствуют реальному формату вызова последних и с формальной точки зрения не могут использоваться в качестве образцов для подражания при написании реальных программ.*

Проблема тупиков, недетерминированность параллельных алгоритмов

Представим себе столовую в парке, по которому прогуливаются беседующие философы. У столовой один вход, около которого стоит дворецкий. Внутри расположен круглый стол на котором по кругу расставлены пять тарелок и положены пять вилок. По центру стола находится большое блюдо спагетти, исправно пополняемое официантом. Сев за стол, очередной философ берет две вилки - свою и любую свободную, и кладет ими спагетти с общего блюда себе в тарелку. Сразу после этого он кладет чужую вилку на место и начинает есть. Поев, он кладет свою вилку на место и уходит. Будем считать, что вилками, положенными на стол после накладывания спагетти или в конце еды, может сразу же воспользоваться любой философ, не принимая во внимание процесс обеспечения чистоты столовых приборов.

Рассмотрим трудности, которые могут возникнуть у философов при таком обслуживании. Основное коварство скрыто во фразе «очередной философ берет две вилки». До тех пор, пока философов меньше 5 вторая вилка по крайней мере для одного из философов найдется. Это значит, что он сможет положить себе спагетти и освободившаяся вилка достанется следующему. Тем самым рано или поздно все сидящие за столом философы получат в свое распоряжение вторую вилку и смогут приступить к еде. Но что произойдет, если за столом окажутся одновременно 5 человек?

Если кто-то из философов к моменту появления пятого человека уже положил себе спагетти и начал есть, то пятый, дождавшись, пока этот кто-то закончит трапезу, получит его вилку и сможет в свою очередь приступить к еде. Но предположим, что к незанятому столу одновременно подошла и села компания из пяти человек, и все одновременно взяли свои вилки. Результат: у каждого по одной вилке, на столе свободных вилок нет. Никто не может положить себе спагетти и

начать есть. В этой ситуации философы останутся навсегда голодными, поскольку вилка освобождается только после того, как человек поест, а приступить к еде в описанной ситуации никто не может, соответственно не может ее и закончить.

Очевидно, что сформулированные правила поведения привели к безвыходному состоянию, которое и называется «тупик» или «клинч». Рассмотрим возможные изменения в правилах, которые позволят нам избежать возникновения тупиков.

Уточним постановку задачи. Будем считать, что человек, получивший в свое распоряжение две вилки, положит спагетти из общего блюда к себе в тарелку и освободит одну из вилок за конечное время. Здесь важно подчеркнуть, что работа должна быть выполнена именно за конечное, неважно большое или нет, но конечное время.

В качестве первого варианта можно предложить дворецкому никогда не пускать к столу больше четырех человек одновременно. При таком подходе, принимая во внимание то, что каждый использует две вилки конечное время, разные философы будут пользоваться второй вилкой поочередно и тупик не наступит. Но у этого решения есть свой недостаток. У дверей столовой возникнет очередь. За последовательность прохождения этой очереди отвечает дворецкий и нет гарантии того, что в ней не окажется философа, стоящего всегда в конце и никогда не попадающего к столу. Такое положение не обязательно является следствием недружелюбного отношения дворецкого к конкретной личности. Известна дисциплина обслуживания по принципу стека – последним пришел – первым пустили. При большом числе философов пришедший первым рискует никогда в столовую не попасть. Если принять дисциплину обслуживания типа «очередь» – по принципу первый пришедший обслуживается первым, возникнут проблемы с обслуживанием "высокоприоритетных" философов, имеющихся в любом коллективе. Таким образом, решив проблему тупика, мы не решили проблему обязательного обслуживания всех философов без исключения. Принимая во внимание третье ключевое требование - любой философ должен быть обслужен за конечное, пусть даже большое, но конечное время, рассмотрим другие возможности организации обслуживания.

Добавим правило, согласно которому если все вилки заняты и никто не ест, все кладут свои вилки на место и через некоторое время повторяют попытку. Если время, через которое философы снова попробуют взять свои вилки будет различным для разных философов, то кто-то из них возьмет две вилки первым и тупика удастся избежать. Несмотря на то, что в повседневной жизни это решение вполне приемлемо: так поступают например, при неожиданном обрыве телефонного разговора, когда абоненты звонят друг другу случайным образом, и кто-то дозванивается, мы не можем принять это решение в качестве

удовлетворительного. Вполне возможно, что философы будут действовать на редкость синхронно, захватывая разделяемый ресурс (вилки) строго одновременно. Эту ситуацию следует расценивать как тупиковую.

Можно добавить правило, согласно которому, если у всех по одной вилке и ни у кого нет на тарелке спагетти, один из философов кладет свою вилку на некоторое время. Требует аккуратности определение того, кто именно из философов должен положить свою вилку. Если это будет тот, кто сидит на фиксированном месте (например на первом), то возможна следующая последовательность действий:

- первый положил вилку на стол;
- второй взял ее, положил себе спагетти, вернул вилку на место;
- второй закончил есть, положил свою вилку на стол и ушел;
- на место второго сел новый человек;
- пришедший философ и первый философ одновременно взяли каждый свою вилку и снова никто не может приступить к еде.

В результате неограниченного повторения этой процедуры будут обслужены все, кроме первого философа. Изменим алгоритм, положив на стол некоторый маркер напротив первого места. Тогда тот философ, перед которым в данный момент лежит маркер должен не только при возникновении ситуации всеобщего ожидания временно предоставить свою вилку другим, но и передвинуть маркер соседу слева. Тогда в следующий раз свою вилку положит уже другой философ. Учитывая, что один из философов может оказаться чрезвычайно медлительным и просто не будет никогда успевать взять положенную кем-то вилку, добавим еще одно ограничение. Можно значительно улучшить алгоритм, разрешив брать положенную на стол вилку только ближайшему слева от нее соседу, на тарелке у которого нет спагетти.

Характерной особенностью параллельных программ является их недетерминированность. Например, мы не можем заранее предсказать последовательность, в которой будут обслужены философы. В описанных выше алгоритмах могут возникать тупиковые ситуации, однако нет оснований утверждать, что они обязательно возникнут за тот или иной промежуток времени, что также свидетельствует о нетедерминированности предложенных алгоритмов.

Разделяемые ресурсы

… если для нас представляют интерес реально работающие системы, то

требуется убедиться, (и убедить всех сомневающихся) в корректности наших построений

… системе часто придется работать в невоспроизводимых обстоятельствах, и мы едва ли можем ожидать сколько-нибудь серьезной помощи от тестов.

Dijkstra E.W.

Эти цитаты из лекций Дейкстры тридцатилетней давности не потеряли своей актуальности и сегодня.

Важнейший вопрос корректной разработки параллельных программ – организация правильного взаимодействия последовательных процессов, обеспечение *синхронизации* между ними. Недооценка роли хорошо спланированной синхронизации взаимодействующих процессов является одной из основных, если не главной, причин появления трудно уловимых логических ошибок. Наличие одной такой ошибки нередко приводит к необходимости пересмотра структуры всей программы. Даже если у каждого процессора системы своя локальная память и в программе явно не используется общая память, остается масса возможностей обращения разных нитей к незащищенным общим ресурсам. Наиболее распространенный случай – использование асинхронно получаемых данных до их реального получения, или модификации асинхронно передаваемых данных до их реальной отправки. Другой пример - обращение нескольких процессов к одному файлу. Несколько более экзотичная ситуация складывается при одновременном вызове из нескольких процессов одного контекста одной и той же нереентерабельной функции, такой как функции печати данных или распределения памяти. Под "нереентерабельной" понимают функцию модифицирующую внутренние статические данные. Например, функция печати данных может использовать внутренний статический буфер для промежуточного хранения выводимой на печать строки. Одновременный вызов такой функции из двух разных процессов, имеющих доступ к общей оперативной памяти приведет к тому, что содержимое строки будет изменятся двумя процессами в процессе ее подготовки, еще до момента ее вывода в файл, что, в свою очередь, приведет к порче данных.

Проблемы, подобные перечисленным, могут быть решены с помощью базовых низкоуровневых средств синхронизации, таких как синхронные каналы связи и семафоры. Однако бесконтрольное, непродуманное использование этих средств может многократно усугубить проблему. Адекватное применение этих средств возможно только на основе их внимательного изучения, к которому мы и переходим.

Выделим два основных способа синхронизации последовательных процессов: с помощью передачи сообщений и с помощью общих переменных. Еще раз подчеркнем, что никаких глобальных переменных, как и никакого «глобального времени», одинакового и доступного для измерения на всех процессорах, в распределенной системе не существует и в связи с этим нельзя говорить о том, что на одном процессоре событие А произошло раньше или позже, чем событие Б на другом процессоре. Вся концепция построения параллельных программ базируется на предположении, что различные процессы могут выполняться с произвольной, неизвестной априори скоростью, вообще говоря, не постоянной во времени. В связи с этим вопросы синхронизации являются определяющими, если мы хотим проектировать устойчивые программы, работоспособность которых не зависит от относительных производительностей процессорных узлов и пропускных способностей используемых каналов связей.

Слабо связанные последовательные процессы

Определим *изолированный последовательный процесс*, как последовательность действий, выполняемых автономно (независимо от окружающей обстановки).

Если двум и более последовательным процессам необходимо взаимодействовать между собой, то они должны быть связаны, то есть иметь средства для обмена информацией. Будем считать, что кроме явных, достаточно редких моментов связи, процессы выполняются совершено независимо друг от друга. Кроме того, будем считать, что такие процессы связаны слабо, подразумевая под этим, что кроме некоторых моментов явной связи, эти процессы рассматриваются как совершенно независимые друг от друга.

Будем называть такие процессы *слабо связанными последовательными процессами.*

Теперь можно определить понятие *параллельной программы*. Под *параллельной программой* будем понимать всю совокупность *слабо связанных последовательных процессов*, запуск которых необходим для решения поставленной задачи. В однопроцессорной системе *программе*, как правило, соответствует один выполняемый процесс - программный модуль, запускаемый под управлением операционной системы на единственном процессоре. Параллельной программе, выполняющейся на наборе процессоров, соответствует ряд *процессов*, запущенных в общем случае на разных процессорах.

Рассмотрим пример, иллюстрирующий проблему разделения ресурсов.

В разных концах комнаты расположены две лампочки, самопроизвольно зажигающиеся случайным образом. Вспыхнувшая лампочка продолжает гореть до тех пор, пока ее не погасят. Двум наблю-

дателям поручено выключать зажигающиеся лампочки и учитывать общее число вспышек, записывая его мелом на доске посередине комнаты.

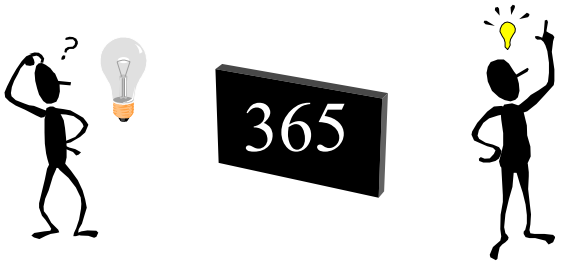

Рис. 42. Подсчет событий двумя наблюдателями

Рассмотрим последовательность действий (рис. 42):

- вспыхнула левая лампочка;
- первый наблюдатель прочел число, написанное на доске (365) и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;
- погасив лампочку и вернувшись, первый исправляет число на доске, записывая получившуюся у него сумму (366).

Результат на доске правильный, если второй наблюдатель на протяжении всего этого процесса к доске не подходил. В противном случае могло получится следующее:

- вспыхнула левая лампочка;
- первый наблюдатель прочел число, написанное на доске (365) и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;
- вспыхнула правая лампочка;
- второй наблюдатель тоже прочел число, написанное на доске (365) и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;
- погасив лампочку и вернувшись, первый исправляет число на доске, записывая получившуюся у него сумму (366);
- погасив лампочку и вернувшись, второй, в свою очередь, исправляет число на доске, записывая получившуюся у него сумму (366).

Результат не совпадает с ожидаемым. Произошло две вспышки, а общее число вспышек, записанное на доске, возросло только на 1! Ошибка наблюдателей очевидно состоит в том, что между чтением первым наблюдателям числа с доски и записью результата, второй наблюдатель успел прочитать еще не исправленное число.

Вероятно, в случае выполнения всех трех действий – чтения числа, прибавления к нему единицы и записи исправленного числа, в виде одной неделимой операции, ошибок возникать не будет.

Критический интервал

Принято называть последовательность действий, выполнение которых должно выглядеть со стороны как одна непрерываемая операция, *критическим интервалом*. Если выделить три операции: чтение числа, прибавления к нему единицы и запись числа в один критический интервал и сделать так, что бы никогда внутри этого интервала не оказывалось больше одного процесса, задача будет решена.

Попробуем построить защиту критического интервала от одновременного вхождения в него двух процессов, ограничившись следующими соглашениями:

1. пусть есть два циклических процесса P1 и P2, каждый из которых может время от времени выполнять некоторую последовательность действий, называемую критическим интервалом;
2. внутри критического интервала каждый процессор находится произвольное, но ограниченное время;
3. вне критического интервала процессы могут находиться сколь угодно долго, вплоть до бесконечности;
4. процессы могут иметь неограниченное число общих переменных;
5. процессы могут читать и записывать значения любых, в том числе общих, переменных без ограничений, в любой момент времени;
6. если два процесса одновременно записывают разные значения (*x* и *y*) в одну и ту же переменную, то в результате ее значение будет равно либо *x* либо *y*, но не их смеси;
7. если один из процессов записывает новое значение в переменную, а второй процесс читает значение этой переменной, то он прочтет либо старое значение переменной, либо то значение, которое туда записал первый процесс;
8. если одному из процессов потребовалось войти в критический интервал, такая возможность должна быть предоставлена ему за конечное время.

Таким образом, неделимыми являются только операции чтения и записи переменных.

Предлагалось довольно много решений этой задачи, и все они оказались некорректными, так что у тех, кто занимался ею, возникло в какой-то момент сомнение, имеет ли она решение вообще.

Dijkstra E.W.

Рассмотрим безопасный, с точки зрения взаимной блокировки, алгоритм.

1. int next=1;

Первый процесс	Второй процесс
2. while (1) *3. {* *4. { произвольные действия }* *5. while(next==2);* *6. { критический интервал }* *7. next = 2;* *8. }*	*9. while (1)* *10. {* *11. { произвольные действия }* *12. while(next==1);* *13. { критический интервал }* *14. next = 1;* *15. }*

Приведенный алгоритм требует, чтобы после того как первый процесс прошел критический интервал, второй процесс непременно также вошел в критический интервал, и наоборот. Действительно, при запуске переменная *next*, показывающая, чья очередь следующим входить в критический интервал, указывает на первый процесс. Это означает, что условие строки 5 неверно и первый процесс войдет в критический интервал первым при условии, что «произвольные действия» в строке 4 будут выполнены за конечное время. Условие в строке 12 верно до тех пор, пока первый процесс не покинет критический интервал и не выполнит оператор строки 7. Теперь переменная *next* указывает, что следующим должен войти в критический интервал второй процесс и условие в строке 5 истинно. Первый процесс не может миновать строку 5, пока переменная *next* не примет значение 1, а единственное место, где она может принять это значение, расположено во втором процессе после критического интервала, что и означает, что пока второй процесс не пройдет критический интервал, первый останется в заблокированном состоянии. По принятым нами условиям, такая ситуация недопустима, поскольку мы разрешаем одному процессу останавливаться на бесконечное время вне критического интервала, но требуем, чтобы другой мог попасть в критический интервал за конечное время.

Единственно возможная последовательность прохождения критического интервала в данном решении – поочередная: 1,2,1,2…, что не решает поставленную задачу.

Ниже приведено первое правильное решение этой задачи, найденное голландским математиком Деккером (*Diekert*) в 1968г. Полное доказательство корректности этого решения можно найти в статье «Взаимодействие последовательных процессов» Э. Дейкстра [8]. Здесь же отметим не тривиальность этого решения, неочевидность его обобщения на большее, чем 2, число процессов. Кроме того, громоздкость всей конструкции в целом ограничивает ценность данного

решения для практического применения. Но приведенное решение хорошо иллюстрирует сложности, возникающие при необходимости ограничения доступа к разделяемым ресурсам – в данном случае к критическому интервалу.

```
int next=1;
int c1=1,c2=1;
```

Первый процесс	Второй процесс
while (1) *{* *{ произвольные действия }* *A1: c1=0;* *L1: if(c2==0)* *{* *if(next==1) goto L1;* *c1=1;* *while(next==2);* *goto A1;* *}* *{ критический интервал }* *next = 2;* *c1 = 1;* *}*	*while (1)* *{* *{ произвольные действия }* *A2: c2=0;* *L2: if(c1==0)* *{* *if(next==2) goto L2;* *c2=1;* *while(next==1);* *goto A2;* *}* *{ критический интервал }* *next = 1;* *c2 = 1;* *}*

Семафоры

В основе проблем, приводящих к столь громоздким средствам обеспечения защищенного доступа к разделяемым ресурсам, лежит односторонний по своей сути доступ к данным в оперативной памяти, используемый традиционными вычислительными системами. Данные либо записываются в оперативную память, либо считываются. Необходимость в более реалистичном и удобном решении, нежели предложено Деккером, привело к созданию Дейкстрой в 1965 концепции семафора. Основная идея семафора состоит в совмещении в единую, неделимую операцию следующих трех операций:

- чтение общей переменной специального вида;
- модификация этой переменной;
- передача управления, в зависимости от исходного значения этой переменной.

Семафор - неотрицательная целая переменная, доступ к которой возможен только с помощью двух неделимых операций, соответственно *Signal* и *Wait*.

Результатом применения операции *Signal* к семафору *S* является увеличение значения семафора на 1. Обратим внимание на то, что

операция *Signal(S)* не эквивалентна операции *S*=*S*+1. Отличие проявляется при одновременном доступе к этой переменной нескольких процессов. Как мы уже видели, если в начале *S*=5, то после выполнения в двух процессах операции *S*=*S*+1 может получиться как *S*=7, так и *S*=6. Выполнение в двух процессах операций *Signal(S)* гарантирует за счет неделимости последних, результат *S*=7.

Результатом применения операции *Wait* к семафору *S* является уменьшение значения семафора на 1, если его значение не становится в результате этого отрицательным. Операция *Wait* как раз и позволяет обеспечить задержку процессов перед доступом к разделяемому ресурсу, если один из процессов уже завладел им. Если процесс выполняет операцию *Wait(S)*, то в случае *S*=0, процесс приостанавливается до тех пор, пока какой либо из процессов не выполнит над семафором операцию *Signal(S)*.

Таким образом может быть приостановлено произвольное количество процессов, выполнивших операцию *Wait(S)* над одним и тем же семафором *S*. После того, как процессор, захвативший ресурс, закончит работу с ним, он должен выполнить операцию освобождения семафора *Signal(S)*. В этот момент значение семафора станет равно 1 и один из приостановленных процессов, в общем случае неизвестно какой именно, сможет закончить операцию ожидания *Wait(S)*, снова уменьшив значение семафора до 0.

Использование семафора дает нам возможность легко и компактно записывать решения задач, аналогичных рассмотренным в этой теме. Например, решение задачи о защите критического интервала запишется теперь в следующем виде:

1. Semaphore Sem;

Первый процесс	Второй процесс
2. while (1) *3. {* *4. { произвольные действия }* *5. Wait(Sem);* *6. { критический интервал }* *7. Signal(sem);* *8. }*	*9. while (1)* *10. {* *11. { произвольные действия }* *12. Wait(Sem);* *13. { критический интервал }* *14. Signal(Sem);* *15. }*

Решение привлекательно не только своей лаконичностью, но и тем, что первый и второй процессы записаны совершенно идентич-

но, что и дает возможность использования данного механизма для управления произвольным числом процессов.

Монитор

В принципе, рассмотренных операций вполне достаточно для защиты разделяемых ресурсов самого разного рода – общих переменных, файлов, каналов связи, нереентерабельных подпрограмм и т.д., однако человек прекрасен еще и тем, что ему свойственно ошибаться. Хорошо, когда вся программ уместилась на четверти листа и единственная защищаемая переменная используется в одном – двух местах. Реальные программы, как правило, несколько больше и разделяемые переменные могут использоваться в них достаточно интенсивно. Необходимость писать каждый раз, когда надо изменить значение такой переменной конструкции вида:

```
Wait(Sem);
модификация разделяемых переменных
Signal(Sem);
```

рано или поздно приведет к тому, что либо *Wait*, либо *Signal*, либо они оба будут забыты в каком-нибудь месте. Для уменьшения вероятности ошибок такого сорта рекомендуется защищать ресурсы не непосредственно с помощью семафоров, а с помощью построенных на их основе мониторов.

Под монитором в данном случае подразумевается набор подпрограмм, инкапсулирующих (включающих в себя) **все** обращения к защищаемому ресурсу. К ресурсу, защищенному с помощью монитора, доступ возможен только с помощью подпрограмм монитора. Монитор, позволяющий устанавливать, изменять и считывать значение переменной *S*, и оформленный в виде отдельного файла может выглядеть следующим образом:

```
1.  static Semaphore Sem;
2.  static int S;
```

Установка значения	Чтение значения	Изменение значения
3. void S_Set (int value) *4. {* *5. Wait(Sem);* *6. S=value;* *7. Signal(Sem);* *8. }*	*9. int S_Get(void)* *10. {* *11. int Stmp;* *12. Wait(Sem);* *13. Stmp=S;* *14. Signal(Sem);* *15. return Stmp;* *16. }*	*17. void S_Plus(int value)* *18. {* *19. Wait(Sem);* *20. S=S+ value;* *21. Signal(Sem);* *22. }*

Использование описателя *static* в строках 1,2 и выделение приведенных подпрограмм в отдельный файл гарантирует невозмож-

ность случайного доступа к переменным *Sem* и *S* иначе, чем через функции монитора *S_Set*, *S_Get* и *S_Plus*, поскольку делает, в соответствии с правилами языка *C,* эти переменные невидимыми в других файлах с исходными текстами программы. Увеличение затрат на написание текста программы с лихвой окупается надежностью получаемого в результате программного кода.

Напоминаем, что реальное название типов данных и формат вызова подпрограмм обработки семафоров PARIX™ отличается от использованного в примере.

Рассмотренные средства предназначены в первую очередь для синхронизации процессов при доступе к данным, расположенным в общей памяти, доступной одновременно нескольким взаимодействующим процессам. Для синхронизации процессов, не имеющих общей памяти, используется второе базовое средство синхронизации процессов и передачи между ними данных – обмен сообщениями, которое будет рассмотрено в последующих разделах.

Тема 4. Языки и средства параллельного программирования

В предлагаемой теме обсуждается ряд вопросов практического построения параллельных программ. Вводятся дополнительные понятия, необходимые для описания компонент параллельной программы.

Между этапом выбора алгоритма, обладающего удовлетворительной степенью параллелизма и получением окончательного ответа решаемой задачи, лежат этапы выбора языка программирования, написания программы, ее отладки, запуска полученного кода на многопроцессорной системе, собственно выполнение расчетов. В ходе выполнения расчетов и по их окончании выполняется определенная работа по интерпретации результатов.

Рассмотрим очередной этап, а именно – выбор языка и средств программирования.

Языки параллельного программирования

Отметим сразу, что для задач общего вида, удовлетворительных средств автоматического порождения параллельных алгоритмов, ориентированных на работу в системах с распределенной памятью, в настоящее время не создано. В общем случае, бремя выбора алгоритма и создания параллельной программы лежит целиком и полностью на ее разработчике. Известные средства создания параллельных программ могут, в лучшем случае, способствовать написанию корректных программ своей наглядностью, лаконичностью и, возможно встроен-

ными формальными средствами проверки соответствия различных частей программы друг другу.

Можно условно разделить языки параллельного программирования на специализированные, содержащие в своем составе операторы и средства описания параллельных процессов, и языки общего применения, к которым добавлены дополнительные библиотеки функций обмена сообщениями между процессами, порождения параллельных процессов, семафоры для организации синхронизации по общим данным и т.д.

Ярким представителем языков первого типа является ***Оккам***. ***Оккам*** является абстрактным языком программирования высокого уровня. Его создание было тесно связано с проектированием и разработкой транспьютерных элементов, что обусловило высокую эффективность выполнения ***Оккам*** программ именно на транспьютерных системах.

Кроме средств, позволяющих описывать последовательное и параллельное выполнение фрагментов программы – элементарных процессов, он содержит в своем составе операторы передачи данных через каналы, операторы альтернативного выполнения элементарных процессов, в том числе процессов ввода/вывода и многое другое. Кроме того, непосредственно в самом языке содержатся конструкции описания физического расположения частей программы на реальных процессорах вычислительной системы. В настоящее время язык Оккам не получил широкого распространения и на рассматриваемых в пособии системах не используется.

Рассмотрим второй подход, согласно которому к традиционному языку программирования типа ***C*** или ***Fortran*** добавляются специальные библиотеки функций.

Отметим, что современные Unix-подобные операционные системы содержат в своем составе средства порождения конкурентно (или параллельно, при наличии в системе нескольких процессоров, объединенных общей памятью) выполняемых процессов. Unix-системы поддерживают механизм обмена данными между процессами с помощью каналов межпроцессорного обмена, интерфейса типа Берклеевских сокетов (Berkeley Sockets), позволяющего пересылать данные между любыми процессами, запущенными на одном или разных компьютерах. Также Unix системы поддерживают достаточно богатые возможности по работе с семафорами и разделяемой памятью. Однако эти средства доступны не на всех параллельных системах и далеко не всегда позволяют полностью использовать возможности, предоставляемые многопроцессорной системой в плане эффективности обмена данными. И, наконец, пожалуй основная причина использования дополнительных, специально разработанных библиотек – обеспечение переносимости создаваемого прикладного обеспечения.

В настоящее время широкое распространение получили два стандарта разработки параллельных программ – PVM (Parallel Virtual Machines – параллельные виртуальные машины) и MPI (Message Passing Interface – интерфейс передачи сообщений), причем MPI является относительно более новым и, по-видимому, более предпочтительным стандартом, активно развиваемым в настоящее время. Известны реализации PVM и MPI практически для всех основных типов операционных систем (в том числе Unix и Windows) и основных типов компьютеров (Sun, HEWLETT PACKARD, Silicon Graphics, IBM PC, и т.д.), что делает разработку программ с использованием этих стандартов особенно привлекательной. Эти стандарты несколько отличаются друг от друга по составу функций и предоставляемым возможностям. На системе Parsytec PowerXplorer частично реализован PVM, на системе ParsytecCC реализованы оба стандарта, но также в ограниченном виде. Эти реализации основаны на специализированном средстве подготовки программ для систем фирмы Parsytec – PARIX™ и поддерживают обмен данными по высокоскоростным (до 40 Мбайт/сек) каналам (HS-Link – High Speed Link). Имеется так же потенциальная возможность использования на системе ParsytecCC общедоступных версий PVM и MPI, использующих для передачи данных локальную Ethernet сеть, но при таком подходе скорость обмена данными не превысит 1 Мбайт/сек, что значительно уступает сети на основе HS-Link.

В связи с этими соображениями дальнейшее изложение посвящено именно системе разработки параллельных программ PARIX™. В качестве базового используется язык высокого уровня *C*. Как подтвердила практика, переход на любую из систем MPI или PVM трудностей не вызывает.

Основные понятия: параллельное и конкурентное выполнение, программа, контекст, нить, канал, семафор

Введем в рассмотрение ряд новых понятий, описывающих части параллельной программы.

Основной единицей описания параллельной программы является последовательный процесс – аналог традиционной подпрограммы языка ***C***. В качестве синонима термина «последовательный процесс» будем использовать термин «ветвь», представляя себе параллельную программу набором одновременно выполняющихся ветвей.

В свою очередь, последовательные процессы могут выполняться последовательно , параллельно или конкурентно между собой. Первые два понятия являются фундаментальными и отражают внутренние свойства программы, определяют правила взаимодействия ее ветвей между собой. Конкурентное выполнение возникает при выполнении двух независимых процессов на одном процессоре в режиме разделения времени. С точки зрения программиста конкурентное вы-

полнение не отличается от параллельного. В принципе, без конкурентного выполнения выполнение процессов можно было бы обойтись совершенно при наличии в вычислительной системе достаточного числа процессоров, объединенных общей памятью. При конкурентном выполнении в каждый момент времени выполняются инструкции только одного из конкурентно запущенных на процессоре процессов. Однако ряд отрицательных эффектов параллельного выполнения – например одновременный доступ к разделяемым, некорректно защищенным от одновременной модификации, переменным, при конкурентном выполнении могут проявляться значительно реже, или не проявляться совсем, проявляясь при истинно параллельном выполнении.

Некоторые из *процессов* отличаются тем, что не имеют доступа к оперативной памяти друг друга. Назовем их, следуя терминологии, принятой в системе программирования PARIX™, *контекстами*.

Итак, каждый *контекст* характерен тем, что он оперирует некоторой оперативной памятью, недоступной другим *контекстам*. Подчеркнем, что два *контекста* могут быть запущены и на одном процессоре, однако они все равно не будут иметь доступа к оперативной памяти друг друга. При начальном запуске *контекстам* (по крайней мере, одному из них) передаются параметры, указываемые пользователем в командной строке, и переменные окружения. Между собой *контексты* могут обмениваться данными по специально организованным каналам связи.

Поскольку контексты оказываются с точки зрения доступной им оперативной памяти, обособленными друг от друга, нет принципиальной разницы, запущены они на одном процессоре, или на разных. Чтобы в дальнейшем не отвлекаться на разницу между контекстами, запущенными на одном и на разных процессорах, введем понятие *виртуального процессора*. Под *виртуальным процессором* будем понимать совокупность аппаратно-программных средств, выделяемых *контексту*. Несмотря на то, что это понятие практически дублирует понятие *контекста*, некоторая разница между ними есть. Она становится понятна, как только мы примем во внимание реальный процесс запуска задачи на счет. В момент запуска контексты получают уникальные идентификаторы, используя которые в качестве адресов, можно передавать данные от одного контекста к другому. В ряде систем программирования эти номера присваиваются только контекстам, запущенным операционной системой при старте программы. Таким образом, приходится различать *контексты,* порожденные системой при старте, и *контексты*, порожденные другими *контекстами* в ходе решения задачи, уже после старта. В начале работы можно указать, сколько именно контекстов следует запустить на каждом реальном процессоре. Именно эти *контексты* получат уникальные идентифика-

торы (как правило, просто номера). В дальнейшем каждый *контекст* может породить новые *контексты*, однако они уже не получат нового адреса в сети межпроцессорной передачи данных, их адрес будет совпадать с адресом, присвоенном их родителю. Виртуальный процессор, таким образом, объединят родительский *контекст* и порожденные им *контексты*. Поскольку с точки зрения программиста нет разницы между реальным и виртуальным процессором, в дальнейшем мы будем использовать термин процессор, имея в виду виртуальный процессор.

Важным свойством *контекста* является возможность порождения *нитей* (иначе называемых *thread – тред, процесс, поток, ветвь*). *Нить* - это *процесс,* который всегда входит в состав одного из *контекстов* и имеет свободный, непосредственный доступ ко всей оперативной памяти, доступной данному *контексту*. *Нить* является частью программного кода, выполняемой *параллельно* или *конкурентно* с любыми другими процессами, в том числе с основным процессом *контекста* или с другими *нитями* этого *контекста*. *Контекст* может породить произвольное, допускаемое конкретной системой число *нитей*, все они будут иметь доступ к общей памяти *контекста*. В ряде систем программирования, например в MPI, не предусмотрено возможности явного запуска *нитей*, однако неявно *нити* используются самой библиотекой MPI, например, при организации операций передачи данных по каналам.

В системах с одним процессором *нити* выполняются конкурентно, и *контексты*, порожденные в ходе выполнения задачи тоже. Так же конкурентно выполняются *контексты*, запущенные в разных виртуальных процессорах на одном физическом процессоре.

Между собой процессы, расположенные на разных процессорах, могут взаимодействовать с помощью обмена сообщениями по специальным каналам обмена данными. В простейшем случае мы будем рассматривать так называемые двухточечные каналы передачи данных. Каждый такой канал соединяет между собой два процесса на одном или на разных процессорах. Вообще говоря, каналы являются однонаправленными, это означает, что для обмена данными между процессами A и B необходимо создать два канала – от A к B, и от B к A, но мы не будем акцентировать на этом внимание, считая каналы двунаправленными, допускающими одновременную передачу информации в обе стороны. Будем полагать, что это так, даже если используемые физические каналы межпроцессорной связи такими свойствами не обладают, перекладывая функции по их обеспечению на соответствующее системное программное обеспечение. С точки зрения программиста, каналы могут быть синхронные и асинхронные, буферизованные и нет, соответствующие свойства будут рассмотрены позже.

Тема 5. Параллельное программирование в системе PARIX™

Рассмотрим характерные вопросы построения параллельных программ на примере задачи сортировки массива.

Последовательная программа сортировки массива

Упорядочить по возрастанию массив n целых чисел на многопроцессорной системе, состоящей из p процессоров (n>>p). Исходный массив сформировать на одном из процессоров, результат разместить на нем же.

Рассмотрим технику построения параллельного алгоритма и оценки его эффективности.

Не ставя своей целью найти эффективное решение, наоборот, ради прозрачности рассуждений, возьмем за основу последовательный алгоритм сортировки массива с помощью метода «пузырька». Метод основан на простом факте: если массив уже отсортирован, то из любых двух соседних его элементов, элемент с меньшим номером не больше элемента с большим номером. Отсюда основная идея метода – последовательно просматриваем массив на предмет наличия неправильно упорядоченных пар. Если такая пара найдена, устанавливаем признак неупорядоченности массива, после чего меняем соседние элементы местами и продолжаем просмотр, меняя при необходимости соседние элементы местами. Если по окончании просмотра признак неупорядоченности массива оказывается установленным, очищаем его и повторяем процедуру. В противном случае массив отсортирован, и можно закончить работу. (Можно показать, что построенная таким образом процедура, действительно закончит свою работу за конечное время, т.е. массив будет упорядочен). Последовательная реализация алгоритма приведена на листинге 1.

Приближенно оценим число операций, выполняемых алгоритмом сортировки. Будем считать за одно действие каждую из операций сравнения двух чисел, перестановки двух чисел и пересылки одного числа с процессора на процессор. В наихудшем случае их общее число определяется как сумма ***n-1*** сравнений и ***n(n-1)/2*** перестановок, в наилучшем (массив изначально отсортирован) - как ***n-1*** сравнение.

```
1.1.  #include <stdio.h >
1.2.  #include <stdlib.h >
1.3.  typedef int  (*FCOMP)(int i, int j);
1.4.  typedef void   FSWAP (int i, int j);
1.5.  void main(int argc,char *argv[]);

1.6.  void mysort(int n, FCOMP comp,
      FSWAP swap)
1.7.  {
1.8.  int i,p;
1.9.  // если по окончании очередного
1.10.    // цикла p==1, то массив
1.11.    // упорядочен

1.12.    for(p=1;p;)
1.13.      {
1.14.      p=0;

1.15.      for(i=0;i<n-1;i++)
1.16.        if(comp(i,i+1)>0)
1.17.          {
1.18.          swap(i,i+1);
1.19.          p=1;
1.20.          }
1.21.      }
1.22.    }

1.23.    static int mas[] =
       {5,3,1,2,4,6,8,4,5,8,2};
1.24.    int comp(int i, int j)
1.25.    {
1.26.    return mas[i]-mas[j];
1.27.    }

1.28.    void swap(int i, int j)
1.29.    {
1.30.    int a;
1.31.    a=mas[i];
1.32.    mas[i]=mas[j];
1.33.    mas[j]=a;
1.34.    }

1.35.    void main(int argc,char
       *argv[])
1.36.    {
1.37.     int n=sizeof(mas)/sizeof(int),i;
1.38.
1.39.    printf("n = %d\n",n);
1.40.     for(i=0;i<n;i++)
1.41.    printf("%2d ",mas[i]);
1.42.     printf("\n");

1.43.     mysort(n,comp,swap);

1.44.     for(i=0;i<n;i++)
1.45.    printf("%2d ",mas[i]);

1.46.    exit(0);
1.47.    }
```

Листинг 1. Последовательная программа сортировки массива методом пузырька

Заметим, что эту задачу в принципе нельзя решить быстрее, чем за ***n/k*** действий, где ***n*** – общее количество чисел, а ***k*** – общее число каналов связи на процессоре, являющимся держателем исходного и результирующего массивов. Действительно, необходим этап распределения чисел по всем процессорам и этап приема результата, а они требуют передачи каждого числа по одному из каналов, откуда и следует приведенная оценка.

При построении параллельного алгоритма будем руководствоваться принципом «геометрического параллелизма» . Назовем процессор, на котором изначально расположен массив сортируемых чисел, «управляющим», остальные – «обрабатывающими». Будем считать, что мы располагаем именно ***p*** обрабатывающими процессорами, причем они пронумерованы от ***1*** до ***p***. Управляющий процессор пусть будет иметь номер ***0***. В соответствии с этим разобьем исходный массив на ***p*** одинаковых частей размером ***n/p*** каждая. Эти части передадим для выполнения на обрабатывающие процессоры, выполним там

алгоритм пузырьковой сортировки, затем соберем отсортированные части на управляющем процессоре и выполним алгоритм их «слияния» в один результирующий массив.

Оценим число операций, необходимое при таком подходе, считая что на управляющем процессоре только один канал связи.

1. Рассылка частей массива по одному каналу: ***p*(n/p) = n*** операций.
2. Параллельная сортировка частей на обрабатывающих процессорах: ***(n/p) (n/p-1)/2*** операций.
3. Сбор отсортированных частей массива: ***p*(n/p) = n*** операций.
4. Слияние: не более ***n*p*** операций. Мы должны ***n*** раз выбрать очередной наименьший элемент из крайних элементов, имеющихся ***p*** отсортированных массивов. Каждый такой выбор можно осуществить не более чем за ***p*** (по числу массивов) действий.

Итого, имеем при параллельной сортировке:

~ ***2n + (n/p)^2/2 +n*p*** операций,

при последовательной сортировке:

~ ***n^2*** операций.

Ожидаемые ускорение и эффективность:

$$S_p = \frac{n^2}{2n + \frac{1}{2}\left(\frac{n}{p}\right)^2 + np} = \frac{2np^2}{2p^3 + 4p^2 + n},$$

$$E_p = \frac{S_p}{p} = \frac{2np^2}{2p^3 + 4p^2 + n}.$$

На рис. 43, 44 приведены графики ускорения и эффективности в зависимости от числа процессоров, соответствующие полученным соотношениям. Число сортируемых точек для каждой кривой постоянно и равно 1024, 2048, 4096, 81192 либо 16384. Можно видеть, что при сортировке массива длиной 16384 на 16 процессорах ускорение превышает 600 (!), что соответствует эффективности распараллеливания 5000% !

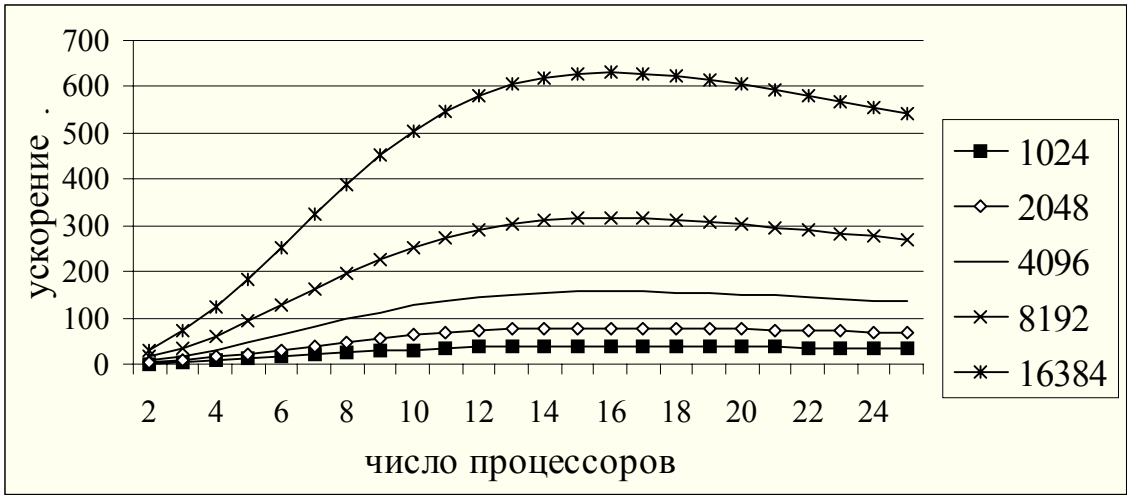

Рис. 43. Ускорение алгоритма сортировки по отношению к простому последовательному алгоритму

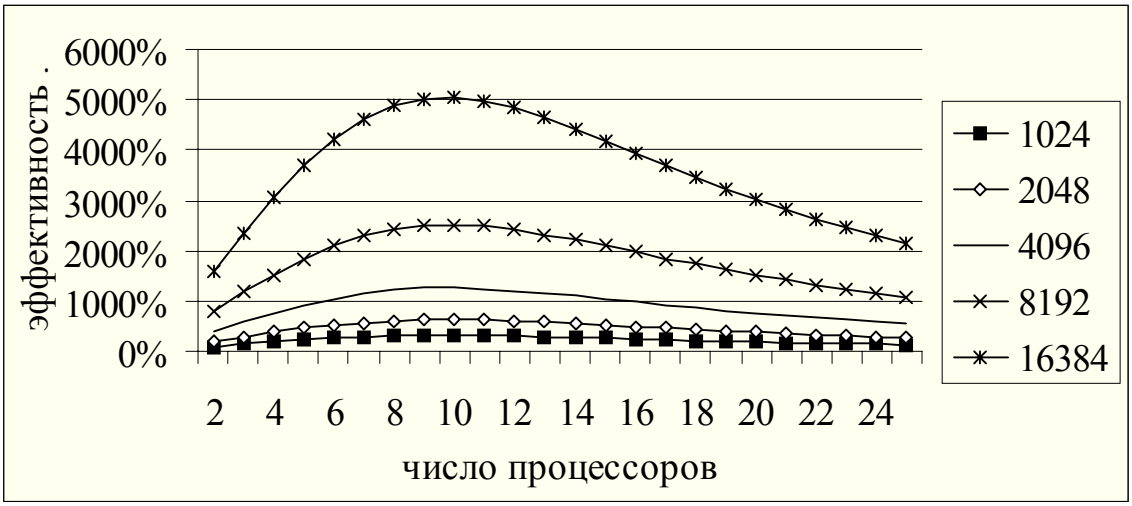

Рис. 44. Эффективность алгоритма сортировки по отношению к простому последовательному алгоритму

Измерение реальных времен выполнения последовательной и параллельной программ дает несколько меньшее ускорение, но все равно намного превышающее число используемых процессоров. Однако, рассматривая закон Амдаля, мы пришли к выводу, что эффективность не может превышать 100%. Проблема заключается в том, что мы выбрали в качестве последовательного алгоритма далеко не самый оптимальный. Попробуем смоделировать выполнение на одном процессоре того же самого параллельного алгоритма. Безусловно, существуют еще более эффективные методы последовательной сортировки, но сейчас нам проще поступить именно так. Разобьем исходный масив на ***p*** одинаковых частей размером ***n/p*** каждая. Эти части отсортируем с помощью алгоритма пузырьковой сортировки поочередно на нашем единственном процессоре, после чего выполним алгоритм их

«слияния» в один результирующий массив. Оценим затраты времени при таком подходе:

1. Рассылка частей массива по одному каналу: 0 операций;
2. Поочередная сортировка частей на одном процессоре: ***p(n/p) (n/p-1)/2*** операций;
3. Сбор отсортированных частей массива: 0 операций;
4. Слияние: не более ***np*** операций;

Итого имеем при параллельной сортировке, по-прежнему:

~ 2n + (n/p)^2/2 +np операций,

при последовательной сортировке предварительно разбитого на части массива:

~ p (n/p)^2/2 +np операций,

Ожидаемые ускорение и эффективность:

$$S_p = \frac{p\frac{1}{2}\left(\frac{n}{p}\right)^2 + np}{2n + \frac{1}{2}\left(\frac{n}{p}\right)^2 + np} = \frac{np + 2p^3}{4p^2 + n + 2p^3},$$

$$E_p = \frac{S_p}{p} = \frac{n + 2p^2}{4p^2 + n + 2p^3}.$$

Соответствующие графики приведены на рис. 45, 46. Теперь результат вполне соответствует ожидаемому. Обращает на себя внимание достаточно быстрый спад эффективности алгоритма при увеличении числа процессоров. Он объясняется сравнительно небольшим числом операций, выполняемых над каждой точкой. Как следствие, затраты на пересылку точки с процессора на процессор становятся с некоторого момента определяющими и снижают ускорение. Показателен также максимум, присутствующий на графиках ускорения. В общем случае оказывается, что для каждой задачи существует определенное число процессоров, на котором эта задача может быть решена быстрее всего. Естественно, что этот максимум сдвигается в сторону большего числа процессоров с увеличением размера задачи (числа точек в сортируемом массиве в данном случае).

Отметим, что несмотря на справедливость наших рассуждений, при решении реальной задачи на конкретной многопроцессорной системе ускорение вполне может превышать число используемых процессоров, а эффективность, соответственно превышать 100%. Как правило, этот эффект проявляется при решении задач, оперирующих большими объемами данных. Попытка удержать большой объем данных на одном процессоре в лучшем случае приведет к увеличению числа обращений к сравнительно медленной оперативной памяти (медленной по сравнению с внутрикристальной кеш-памятью процессора), в худшем – к выгрузке операционной системой части данных из оперативной памяти на диск, и интенсивным обменам с медленной

дисковой памятью (процесс, называемый swapping, своппирование данных). При решении той же задачи на нескольких процессорах, начиная с некоторого их числа своппирования уже не происходит. Более того, при большом числе процессоров все часто используемые данные могут оказаться размещенными именно в быстрой кеш-памяти процессоров, что приводит к резкому увеличению скорости выполнения операций. Эта ситуация эквивалентна дополнительному эффективному увеличению производительности каждого процессора и всей системы в целом, что, естественно сокращает время обработки задачи. Подчеркнем еще раз, что этот эффект возникает не вследствие проявления внутренних свойств параллельных алгоритмов, но благодаря чисто конструктивным особенностям современных процессоров.

Рассмотрим соответствующую программу, ориентированную на работу под управлением системы PARIX™.

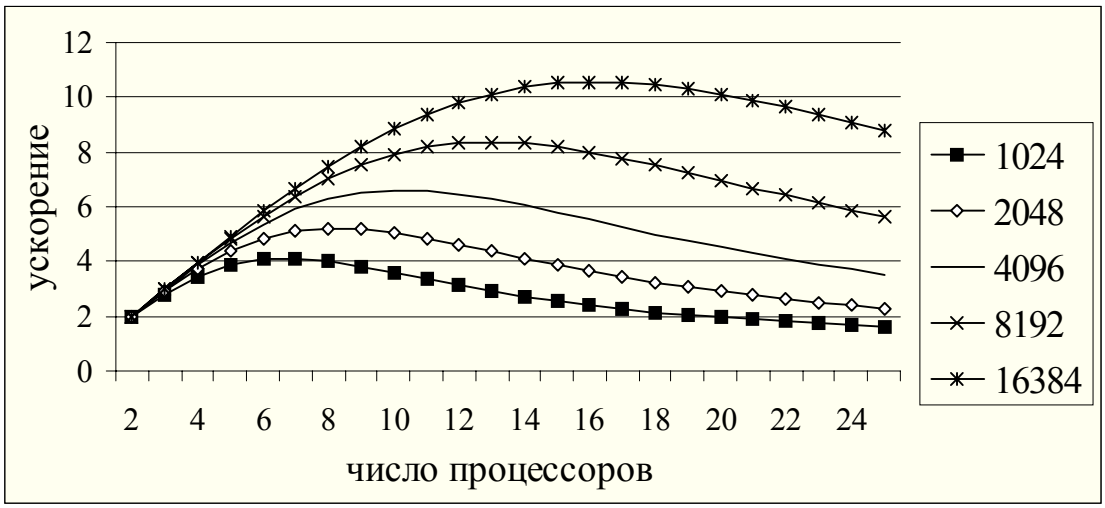

Рис. 45. Ускорение алгоритма сортировки по отношению к блочному последовательному алгоритму

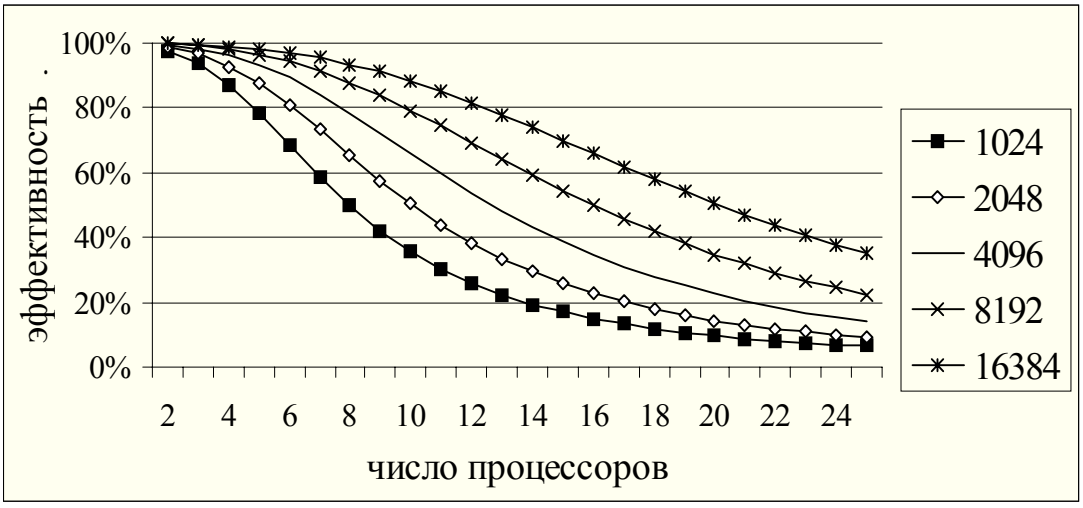

Рис. 46. Эффективность алгоритма сортировки по отношению к блочному последовательному алгоритму

Практика построения параллельных программ

Система PARIX™ предоставляет достаточно широкие возможности в плане выбора средств для реализации той или иной программы. Во многих случаях поставленная цель может быть достигнута с помощью различных средств. Выбор наиболее эффективного метода для каждого конкретного случая представляет определенные трудности. Вместе с тем некоторые приемы построения программ уже хорошо себя зарекомендовали и предлагаются в качестве удобного и вполне приемлемого подхода.

Укажем наиболее характерные моменты, допускающие неоднозначную интерпретацию.

Ввод и вывод исходных данных

Ввод и вывод исходных данных с внешних дисковых накопителей может осуществляться как минимум двумя разными способами:

- Ввод/вывод данных выполняет один из процессоров, называемый в дальнейшем «управляющий». После ввода данных управляющий процессор передает данные по высокоскоростным каналам связи остальным процессорам (назовем их «удаленными»). Для вывода данных, все процессоры передают результаты управляющему процессору, осуществляющему их непосредственный вывод на диск;
- Ввод/вывод данных выполняется непосредственно каждым процессором.

Преимуществом первого подхода является его универсальность. Он применим даже для тех систем, в которых удаленные процессоры не имеют непосредственно доступа к дискам вычислительной системы, с которой осуществлен запуск. Вторым доводом в пользу использования первого подхода на ряде систем, может служить, как ни странно, меньшее время записи данных, по сравнению со вторым методом. Например, в системе Parsytec CC доступ удаленных процессоров к дисковым накопителям, установленным на так называемом «мастер»-узле, осуществляется по медленной Ethernet сети, на порядок уступающей в пропускной способности высокоскоростной сети HS-Link, тогда как запись на диск непосредственно с процессора «мастер»-узла осуществляется максимально быстро (рис. 47).

Наряду с преимуществами у первого подхода есть некоторые недостатки. Основной из них проявляется при обработке данных, объем которых превышает оперативную память управляющего процессорного узла. В этой ситуации необходимо выполнять запись на диск непосредственно после получения данных от каждого из удаленных процессоров. Это ограничение затруднит использование асинхронного сбора данных с удаленных процессоров, но учитывая, что обмен с

диском, как правило, сравнительно редкая операция, ради надежности и универсальности в ряде случаев можно поступиться некоторой потерей эффективности.

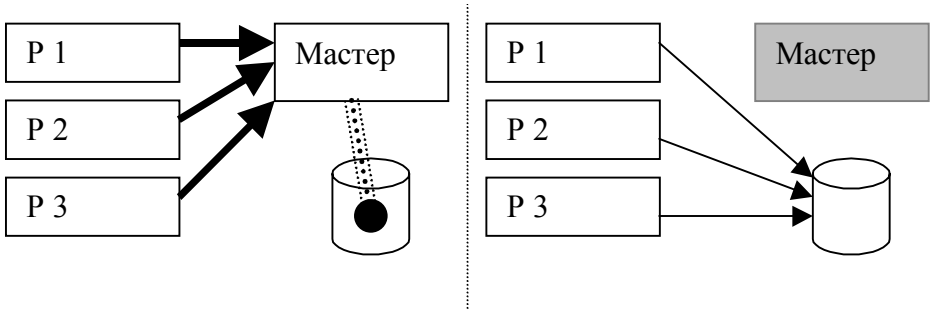

Рис. 47. Два способа организации дискового ввода/вывода

Рассмотрим теперь второй подход. При раздельной записи данных на диск каждым процессором возможны, по крайней мере, два варианта:

a) запись всеми процессорами одного большого файла - каждый процессор пишет свой фрагмент этого файла;
b) запись каждым процессором отдельного файла – общее число файлов при этом соответствует числу процессоров.

Вариант a) вызывает, как минимум, два возражения. Первое: на практике при одновременном обращении для записи к одному файлу большого числа процессов, могут возникать конфликты доступа. В лучшем случае эти конфликты просто снижают производительность системы, в худшем – приводят к плохо локализуемым ошибкам в файлах результатов. Второе: при обработке больших объемов данных возникает желание записывать на диск данные в сжатом виде. При совместной записи одного файла каждый процессор должен уметь определять ту область файла, которая принадлежит именно ему. Этому условию легко удовлетворить, если из исходных данных можно априори однозначно определить объем данных, записываемый каждым процессором. Однако сложно до выполнения сжатия предсказать, какой объем займут те или иные данные после сжатия, что заметно усложняет процедуру записи.

Вариант b) может быть принят в качестве удовлетворительного при условии что задача всегда запускается на одном и том же числе процессоров. Для задач моделирования сложных многомерных физических процессов характерное время расчета одной задачи исчисляется неделями. В этих условиях задачу периодически останавливают и снова запускают. Хорошо написанная программа должна работать на любом, имеющемся в данный момент числе свободных процессоров, то есть быть масштабируемой. Оправданием отказа от запуска на предоставленном числе процессоров может служить недостаток общей оперативной памяти на выделенной группе процессоров, их низкая суммарная производительность, но никак не то, что при предыдущем

запуске было использовано, не совпадающее с предоставляемым теперь, число процессоров. Если при первом запуске программы, подготовленном по методу b) было использовано 10 процессоров, то и записано будет 10 файлов. В этом случае при попытке запуска программы во второй раз на 15 процессорах, неясно, какие данные должны считывать последние 5 процессоров.

Исходя из изложенных соображений, рекомендуется локализовывать все обмены с диском на управляющем процессоре.

Виды каналов связи

Для обмена данными между процессами мы будем использовать двухточечные каналы передачи данных. Данные могут передаваться *синхронно* или *асинхронно*. В свою очередь, асинхронная передача данных может быть *буферизованной* или *небуферизованной*. Рассмотрим подробнее эти понятия.

При синхронной передаче данных передающий (или принимающий) процесс, начав операцию обмена данными, приостанавливает свою работу до тех пор, пока соответствующий ему принимающий (или передающий) процесс не будет готов и также не приступит к обмену данными.

При асинхронной передаче данных, как передающий, так и принимающий процессы инициируют операцию обмена и немедленно продолжают свое выполнение, вне зависимости от того, готов соответствующий принимающий или передающий процесс к обмену или нет. Реальная передача данных и выполнение затребовавших ее процессов, может происходить таким образом одновременно. Для того, что бы убедится, в том, что данные реально переданы (приняты) предусматриваются дополнительные средства, рассматриваемые ниже.

Таблица 3

Каналы обмена данными

Способ установки связи	Синхронный обмен данными	Асинхронный обмен данными
Виртуальные каналы ***virtual link***	Send SendLink Recv RecvLink Select GetLinkCB	Asend Arecv AInit Aexit ASync AWait Ainfo могут использоваться буфера на приемном и передающем конце
Произвольные ***(random)*** коммуникации - без установления каналов, при сообщениях < 1024 байт. *MAX_MESSAGE_SIZE*	SendNode Использует RecvNode ***PutMessage*** ***GetMessage*** ***ExchangeMessage*** Или Виртуальные каналы	[User] PutMessage [User] GetMessage [User] ExchangeMessage

Библиотека PARIX™ предоставляет несколько режимов обмена данными между процессорами. Соответствующие им подпрограммы перечислены в таблице 3.

Допускается передача данных без предварительного создания виртуального канала передачи данных или с созданием такого канала. Первый метод не рекомендуется к использованию и, на современных системах поддерживается не полностью. Соответствующие ему подпрограммы перечислены во второй строке таблицы **3** и нами рассматриваться не будут. Для обмена данными рекомендуется использовать процедуры передачи данных через виртуальные каналы данных.

Пользователь может самостоятельно создавать *виртуальные* или *локальные каналы* для передачи данных между процессами, расположенными на разных или на одном процессоре, соответственно. В дальнейшем будем называть их просто *каналами*. Для создания *виртуального канала* используется функция:

LinkCB_t *ConnectLink (int Processor, int RequestId, int *Error);

Для создания и уничтожения *локального канала* используются функции:

int LocalLink (LinkCB_t *Link[2]);
int BreakLink (LinkCB_t *Link);

Виртуальные и локальные каналы могут быть объединены в виртуальные топологии с помощью функций:

int NewTop (int nLinks);
int AddTop (int TopId, LinkCB_t *Link);
int AddTop_Data (int TopId, void *Data);

Для получения информации о виртуальной топологии или для ее уничтожения используются функции:

void *GetTop_Data (int TopId, int *Error);
LinkCB_t *GetLinkCB (int TopId, int LogLinkId, int *Error);
int FreeTop (int TopId);

В некоторых специфических случаях указанные средства действительно бывают необходимы, однако для большинства приложений удобно использовать одну из стандартных виртуальных топологий . Некоторые из стандартных виртуальных топологий приведены в таблице 4.

Таблица 4

Каналы обмена данными

MakePipe	***Линейка***	***Make2Dtorus***	***Двумерный тор***
MakeRing	***Кольцо***	***Make3Dtorus***	***Трехмерный тор***
MakeStar	***Звезда***	***MakeHCube***	***Гиперкуб***
Make2Dgrid	***Двумерная решетка***	***MakeTree***	***Дерево***
Make3Dgrid	***Трехмерная решетка***	***MakeClique***	***Клика***

Можно было бы воспользоваться топологией типа *Звезда*, но в иллюстративных целях использована топология типа *Клика*, применимая в большинстве случаев. Для разделения различных информационных потоков можно создать в программе несколько виртуальных топологий одного или разных типов.

Создание виртуальной топологии

Для создания топологии *Клика* можно использовать такую последовательность действий (листинг 2):

```
2.1.    int             reqId = 0;
2.2.      int           topId;
2.3.      CliqueData_t      *cliqueData;
2.4.      int           size = GET_ROOT()->ProcRoot->nProcs;

2.5.      topId = MakeClique (reqId, size,
2.6.            MINSLICE, MAXSLICE,
2.7.            MINSLICE, MAXSLICE,
2.8.                  MINSLICE, MAXSLICE);
2.9.      if (topId < 0) { printf("Error after MakeClique\n"); return; }
2.10.     cliqueData = GetClique_Data (topId);
```

Листинг 2. Создание топологии типа Клика

В строке 2.1 (листинг 2, строка 1) задается некоторое произвольное, одинаковое для всех процессоров, участвующих в создании виртуальной топологии, число. При создании нескольких топологий необходимо указывать разные числа для разных топологий.

Строки 2.2 и 2.3 описывают идентификатор топологии и информационную структуру соответствующих ей данных (листинг 3).

В строке 2.4 определяется общее число процессоров, доступных программе.

В строке 2.5 создается топология. В параметрах подпрограммы указывается размер топологии (он не должен превышать общего числа процессоров в системе) и границы группы процессоров, участвующих в создании топологии. Этот вызов производится на всех процессорах указанной группы. Поскольку все процессоры по соглашению, принятому в системе, пронумерованы соответственно узлам трехмерной решетки, задается 6 параметров, определяющих диапазоны используемых узлов по осям X,Y и Z, соответственно. MINSLICE,

MAXSLICE – это предопределенные константы, указывающие на желание пользователя включить в процесс построения топологии все доступные процессоры. Поскольку в направлении Z всегда используется только одна плоскость, задавать некоторые отличные от (MINSLICE, MAXSLICE) можно по осям X и Y. Например, вызов:

```
topId = MakeClique (reqId, 7,
                    1, 4,
                    1, 2,
                    MINSLICE, MAXSLICE);
```

приведет к построению топологии, содержащей 7 из 8-и, выделенных на рис. 48, процессоров.

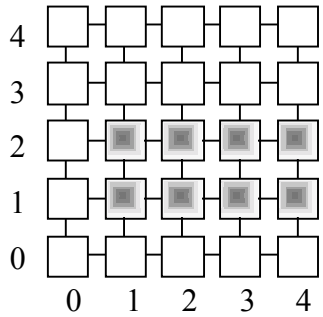

Рис. 48. Пример топологии, заданной на неполном наборе процессоров

В строке 2.9 производится проверка успешности создания топологии. В случае ошибки печатается сообщение и выполнение программы прерывается.

В строке 2.10 адрес информационной структуры данных, связанной с созданной топологией заносится в переменную *cliqueData*. Таким образом получен дескриптор, определяющий топологию – *topId*, и адрес ее информационной структуры данных (листинг 3) – *cliqueData*.

Связанная с топологией *Клика* информационная структура данных имеет следующий формат:

```
3.1.   struct CliqueData_t {
3.2.            char type; /* имя топологии, равно CLIQUE_TYPE */
3.3.            int  status; /* статус процессора */
3.4.            int  id; /* номер процессора */
3.5.            int  size; /* общее число процессоров в топологии */
3.6.        };
```

Листинг 3. Информационная структура топологии типа Клика

Здесь:

status - статус процессора, равен CLIQUE_IN, если процессор принадлежит топологии, в противном случае равен CLIQUE_NONE, что может быть в случае формирования топологии меньшего размера, нежели указано процессоров при ее создании;

size - общее число узлов в топологии, задаваемое при создании топологии. Оно не может быть больше выделенного задаче общего количества процессоров;

id - номер процессора, лежащий в интервале от 0 до ***size***-1**.**

Через виртуальные каналы построенной виртуальной топологии можно синхронно передавать данные. Например, для передачи массива *mas* из 10 целых чисел процессору *i* достаточно выполнить следующий вызов:

```
int mas[10];
Send (topId, i, mas, 10*sizeof(int));
```

Для приема этого же объема данных от процессора *j* достаточно выполнить следующий вызов:

```
Recv (topId, j, mas, 10*sizeof(int));
```

При необходимости асинхронной передачи данных через созданную топологию следует предварительно инициализировать ее с помощью функции *AInit* и использовать функции *ASend* и *ARecv* для обмена данными по ее каналам связи.

Параллельная программа сортировки массива

Теперь можно перейти непосредственно к построению параллельной программы сортировки массива (листинг 4). Рассмотрим возможную структуру программы, использующей синхронную модель обмена данными:

- Создание виртуальной топологии;
- Вызов управляющего модуля на процессоре 0 и обрабатывающих модулей на остальных процессорах топологии.

В управляющем модуле:

- Формирование массива исходных данных;
- Передача каждому из обрабатывающих процессоров числа предназначенных ему элементов массива и соответствующего фрагмента исходного массива;
- Прием отсортированных фрагментов массива;
- Выполнение операции слияния полученных фрагментов в один массив;
- Вывод результата на печать;
- Конец работы.

В обрабатывающем модуле:

- Прием от процессора 0 числа элементов в принимаемом массиве;
- Прием от процессора 0 заданного числа элементов массива

- Последовательная сортировка полученного массива методом пузырька
- Передача отсортированного массива на процессор 0;
- Конец работы.

4

Не останавливаясь подробно на вопросах проверки корректности полученного результата, отметим, что проверка на упорядоченность тривиально реализуется с помощью последовательного алгоритма. Проверку на совпадение исходного и полученного множеств чисел легко реализовать, используя во время сортировки дополнительный массив ссылок. В начале этот массив содержит номера всех чисел. Переставляя эти номера, вместо самих элементов можно выполнить сортировку. Тогда по окончании работы достаточно убедиться, что массив ссылок содержит все числа от 0 до n-1 ровно один раз.

```
4.1.  include<stdlib.h>
4.2.  #include<stdio.h>
4.3.  #include<sys/root.h>
4.4.  #include<sys/time.h>
4.5.  #include<virt_top.h>

4.6.  typedef int  (*FCOMPLOCAL)(int *mas, int i, int j);
4.7.  typedef void   FSWAPLOCAL (int *mas, int i, int j);

4.8.  void main(int argc,char *argv[]);
4.9.  void mysortLocal(int *mas, int n, FCOMPLOCAL comp, FSWAPLOCAL swap);
4.10. int comp(int *mas, int i, int j);
4.11. void swap(int *mas, int i, int j);

4.12. int             topId;
4.13. int             reqId = 0;
4.14. CliqueData_t  *cliqueData;
4.15. void master(void);
4.16. void slave(void);
4.17. #define MPID (GET_ROOT()->ProcRoot->MyProcID)
4.18. #define CLID (cliqueData->id)

4.19. void main(int argc,char *argv[])
4.20. {
4.21.   int  size = GET_ROOT()->ProcRoot->nProcs;

4.22.    printf("MPID%d: Hello\n",MPID);

4.23.    topId = MakeClique (reqId, size,
4.24.          MINSLICE, MAXSLICE, MINSLICE, MAXSLICE, MINSLICE,
      MAXSLICE);

4.25.    if (topId < 0)
4.26.          {
```

```
4.27.           printf("MPID%d: Clique not created\n",MPID);
4.28.           AbortServer(1);
4.29.           }
4.30.      else
4.31.       {
4.32.           cliqueData = GetClique_Data (topId);
4.33.           if (cliqueData->status == CLIQUE_NONE)
4.34.    printf("MPID%d: Processor out of Clique \n",MPID);
4.35.           else
4.36.                   {
4.37.                   printf ("MPID%d: Clique of size %d, ив=%в\n",
4.38.                           MPID, cliqueData->size, CLID);
4.39.                   if(CLID == 0) master();
4.40.                   else            slave();
4.41.                   }
4.42.    exit(0);
4.43.   }

4.44.   void master(void)
4.45.   {
4.46.    int i,k;
4.47.    int start=1;
4.48.    int nslave = cliqueData->size-1;
4.49.    int *mas;
4.50.    int **wrk;
4.51.    int *iwrk;
4.52.    int m=100;  // cells by processor
4.53.    int n=m*size; // all cells
4.54.    int sorttime;

4.55.   mas=(int*)malloc(n*sizeof(int));

4.56.    wrk=(int**)malloc(nslave*sizeof(int*));
4.57.    for(i=0;i<nslave;i++)
4.58.           wrk[i]=(int*)malloc(m*sizeof(int));

4.59.   iwrk=(int*)malloc(nslave*sizeof(int));

4.60.    srand(n);
4.61.    for(i=0;i<n;i++) mas[i]=rand();
4.62.    printf("Before  sorting:\n");
4.63.    for(i=0;i<n;i++)
4.64.     {
4.65.      printf(" %d ",ar[i]);
4.66.      if (i%4==3) printf("\n");
4.67.     }
4.68.    printf("\n");

4.69.    sorttime=TimeNow();
4.70.   for(i=0;i<nslave;i++)
4.71.     {
4.72.     Send (topId, i+1, &m, sizeof(int));
```

```
4.73.      Send (topId, i+1, mas+i, m*sizeof(int));
4.74.      }
4.75.    for(i=0;i<nslave;i++)
4.76.      Recv (topId, i+1, wrk[i], m*sizeof(int));

4.77.   for(i=0;i<nslave;i++) iwrk[i]=0;

4.78.     for(i=0;i<n;i++)
4.79.            {
4.80.            for(j=0,jmin=0;j<nslave;j++)
4.81.            if(wrk[jmin][iwrk[jmin]]>
4.82.              wrk[j][iwrk[j]])jmin=j;
4.83.            mas[i]=wrk[jmin] [iwrk[jmin]];
4.84.            iwrk[jmin]++;
4.85.            }

4.86.     sorttime=TimeNow()-sorttime;
4.87.    printf("Sorttime = %d\n",
4.88.            sorttime/1000000);
4.89.    printf("After  sorting:\n");
4.90.     for(i=0;i<n;i++)
4.91.      {
4.92.       printf(" %d ",ar[i]);
4.93.       if (i%4==3) printf("\n");
4.94.      }
4.95.   }

4.96.   void slave(void)
4.97.   {
4.98.     int n;
4.99.     int *mas;
4.100.
4.101.  Recv (topId, 0, &n, sizeof(int));
4.102.  mas=(int *)malloc(n*sizeof(int));

4.103.  Recv (topId, 0, mas, n*sizeof(int));
4.104.  mysortLocal(mas,n,comp,swap);
4.105.  Send(topId,0,mymas,n* sizeof(int));
4.106.  }

4.107.  void mysortLocal (int *mas, int n, FCOMP comp, FSWAP swap)
4.108.  {
4.109.  int i,p;
4.110.   // если по окончании очередного
4.111.  // цикла p==1, то массив упорядочен
4.112.
4.113.   for(p=1;p;)
4.114.     {
4.115.     p=0;
4.116.
4.117.     for(i=0;i<n-1;i++)
4.118.       if(comp(mas,i,i+1)>0)
```

```
4.119.      {
4.120.      swap(mas,i,i+1);
4.121.      p=1;
4.122.      }
4.123.    }
4.124.  }

4.125.  int comp(mas,int i, int j)
4.126.  {
4.127.  return mas[i]-mas[j];
4.128.  }

4.129.  void swap(mas,int i, int j)
4.130.  {
4.131.  int a;
4.132.  a=mas[i];
4.133.  mas[i]=mas[j];
4.134.  mas[j]=a;
4.135.  }
```

Листинг 4. Параллельная программа сортировки массива

Тема 6. Компиляция и выполнение параллельных программ под управлением PARIX™

Использование справочной системы

Для получения подробной информации об упоминаемых в пособии команд и подпрограмм PARIX™ можно использовать команду:

px man <имя команды | имя библиотечной функции>

Приведем краткий список наиболее употребляемых команд PARIX™ и UNIX с их кратким описанием:

Префикс	Команда	Описание
px	***ancc***	компилятор с языка C, C++
px	***f77***	компилятор с языка Fortran-77
px	***nrm***	утилита управления ресурсами многопроцессорной системы
px	***run***	утилита запуска параллельных программ на выполнение
px	***man***	получение справки о командах и библиотечных функциях PARIX™
	man	получение справки о командах, библиотечных функциях и системных вызовах UNIX
	ps	получение списка активных процессов UNIX
	kill	прекращение выполнения программы

Префикс	Команда	Описание
	telnet	установление связи с удаленной системой
	ftp	передача файлов на удаленную систему
	make	утилита автоматизации сопровождения проектов

Компиляция, сборка и запуск программы на выполнение

Для компиляции программных модулей используется команда

px ancc <имя файла.c> -c -qcpluscmt –O3

Здесь:

-c - указание транслятору выполнить только компиляцию файла, не создавая выполняемый файл;

-qcpluscmt - разрешение использования в тексте программ комментариев, начинающихся с "//" - в стиле C++;

-O3 - максимальный уровень оптимизации программы. При оптимизации этого уровня компилятор может, для достижения большей скорости выполнения программы, менять последовательность выполнения операторов программы. Не исключено, что в результате таких действий программа будет выполняться некорректно. Как правило, этого не происходит, однако компилятор выдает соответствующее предупреждение. В связи с этим при отладке рекомендуется использовать оптимизацию второго уровня (***-O2***).

Для сборки программы используется команда

px ancc <имя файла1>.o ... <имя файла n>.o [опции] –o <имя создаваемого выполняемого файла программы>

Для запуска программы на выполнение используется команда

***px run –a <имя группы процессоров> ***
<имя программы> [<аргументы программы>]

или

***px run –a <имя группы процессоров> <число виртуальных процессоров> ***
<имя выполняемого файла программы> [<аргументы программы>]

Здесь:

<имя группы процессоров> - групповое или уникальное имя раздела процессоров, на котором следует запустить программу;

<число виртуальных процессоров> - одно или два числа - n_x n_y, определяющие размер виртуальной линейки или решетки процессоров на которой следует запустить программу;

Все процессоры систем Parsytec CC нумеруются подряд целыми числами, начиная с 0. Для программиста или пользователя система представляется линейкой процессоров вне зависимости от типа использованной реальной конфигурации. Задаче пользователя может быть выделена любая группа идущих подряд процессоров. Группы процессоров (разделы) имеют индивидуальные и групповые имена.

Групповые имена имеют вид ***pp****nn*, где *nn* – число требуемых процессоров. Например, команда

px run –a pp10 test.px

запустит программу ***test.px*** на любом свободном 10-процессорном разделе.

Индивидуальные имена имеют вид ***cc****nn_mm*, *nn* – число процессоров в разделе, *mm* – номер первого процессора раздела. Например раздел с именем ***cc***10_5 содержит 10 процессоров и начинается с процессора номер 5.

Информацию о имеющихся в системе разделах можно получить с помощью команды

px nrm -pp

В ее выдаче присутствуют строки, содержащие индивидуальные и групповые имена разделов:

индивидуальные имена ↓ **групповые имена** ↓

Partition cc1_9: attributes = (pp1), corner atoms = (9,0,0) (9,0,0)
Partition cc1_8: attributes = (pp1), corner atoms = (8,0,0) (8,0,0)
Partition cc2_0: attributes = (pp2), corner atoms = (0,0,0) (1,0,0)
Partition cc2_1: attributes = (pp2), corner atoms = (1,0,0) (2,0,0)
Partition cc2_2: attributes = (pp2), corner atoms = (2,0,0) (3,0,0)

Информацию о имеющихся в системе свободных разделах можно получить с помощью команды

px nrm -pa | grep *

В ее выдаче присутствуют строки, содержащие уникальные имена свободных разделов:

cc1_11 | * | Mon Jun 28 23:19:43 1999

```
cc1_10   | *       | Wed Jun 16 20:45:59 1999
cc2_10   | *       | Mon Jun 28 23:20:26 1999
```

Для проверки состояния запущенной программы удобно использовать команду

nrf

Ее выдача имеет следующий вид:

```
1.  -------------------------
2.  user      pts/0      Jun 12 16:33    (imm8)
3.  -----
4.  cc002259   up 12 days, 10:15,   load average:  0.96,  0.85, 0.86
5.  cc001741   up 12 days, 10:15,   load average:  0.96,  0.86, 0.88
6.  cc002253   up 12 days, 10:15,   load average:  0.98,  0.93, 0.93
7.  cc002231   up 12 days, 10:15,   load average:  0.45,  0.63, 0.75
8.  cc002263   up 12 days, 10:15,   load average:  1.00,  0.98, 0.95
9.  cc002262   up 12 days, 10:15,   load average:  1.00,  0.92, 0.84

10. cc002261   up 12 days, 10:15,   load average:  0.99,  0.91, 0.86
11. cc002265   up 12 days, 10:15,   load average:  0.99,  0.91, 0.90
12. cc002858   up 12 days, 10:15,   load average:  0.96,  0.85, 0.82
13. cc001439   up 12 days, 10:15,   load average:  0.43,  0.82, 0.87

14. IMM7    up 12 days, 10:14,    load average:  0.12, 0.08, 0.09
15. IMM7a    up 12 days, 10:15,    load average:  0.00, 0.00, 0.00

16. A        :  user@IMM7
17. AAAAAAAAAA**
18. -----
19. -----
20. user 23936 22084   0   Jun 04  pts/3 27:56 run -a cc10_0 10 /export/ext2/user
21. -----
```

Интерес представляют строки 4–13, соответствующие разделу ***cc10_0***, указанному в строке 20. Выделенная колонка показывает интенсивность использования программой процессорного времени соответствующего процессора. В данном примере процессоры с 0 по 3 и с 4 по 8 загружены практически полностью, процессоры 4 и 9 частично простаивают (примерно половину времени), возможно из-за несбалансированности вычислительной нагрузки. Процессоры 10 и 11 свободны и могут быть использованы для запуска других параллельных задач.

Для остановки запущенной программы можно использовать нажатие клавиш ***<CTRL-C>*** или команду ***kill***. В качестве аргумента команды ***kill*** надо указать номер процесса ***run***, выделенный в строке 20 увеличенным шрифтом:

kill 23936

Утилита *make*

Для уменьшения числа набираемых в ходе работы команд удобно использовать утилиту ***make***. Пусть исходные тексты программ расположены в двух файлах ***sub1.c*** и ***sub2.c*** и они используют заголовочный файл ***share.h***. Выполняемый файл назовем ***prog.px.***

Последовательность команд, выполняющих компиляцию, сборку и запуск на выполнение полученной программы, могла бы выглядеть следующим образом:

```
5.1.  px ancc -c sub1.c
5.2.  px ancc -c sub2.c
5.3.  px ancc sub1.o sub2.o - o prog.px
5.4.  px run -a pp10 prog.px
```

Листинг 5. Компиляция, сборка и запуск программы на 10 процессорах системы Parsytec CC

Для выполнения действий листинга 5 с помощью единственной команды ***make*** необходимо подготовить управляющий файл (назовем его ***makefile,*** листинг 6).

```
6.1.  prog.px: sub1.o sub2.o
6.2.   <Tab> px ancc sub1.o sub2.o  -o $@
6.3.
6.4.  sub1.o: sub1.c share.h
6.5.  sub2.o: sub2.c share.h
6.6.
6.7.  .c.o:
6.8.  <Tab> px ancc -c $(opt601c) -o $@ $*.c
6.9.
6.10. r2: prog.px
6.11. <Tab> px run -a pp2 prog.px
```

Листинг 6. Пример управляющего файла *makefile* для утилиты *make*

В строке 6.1 указано, что для создания выполняемого файла ***prog.px*** необходимо иметь объектные файлы ***sub1.o*** и ***sub2.o***.

В строке 6.2 указано, с помощью какой именно команды следует создать выполняемый файл. $@ - встроенная переменная, соответствующая имени целевого файла, записанного перед ":" в строке зависимостей 6.1 (в данном случае это имя ***prog.px***). Обратим внимание, что все команды, указанные в управляющем файле обязательно начинаются с символа табуляции ***<Tab>***. Таким образом, если объектные файлы ***sub1.o*** и ***sub2.o*** уже существуют, причем созданы после того, как соответствующие исходные файлы ***sub1.o*** и ***sub2.o*** были последний раз модифицированы, выполненная команда будет полностью совпадать с 5.3.

В строках 6.4, 6.5 после ":" указаны файлы "источники", то есть те исходные и заголовочные файлы, от которых "зависят" объектные файлы. В строках 6.6, 6.7 указано, как именно следует создавать объектные файлы. ***$**** - это встроенная переменная, соответствующая именно имени (без расширения) целевого файла. В случае, если после последней трансляции исходные тексты или заголовочный файл были модифицированы, после запуска утилиты ***make*** будет выполнена одна (или обе) из команд 5.1, 5.2, а потом команда 5.3.

Если необходимо не только скомпилировать исходные тексты и подготовить выполняемый файл, но и запустить программу, вместо команды ***make*** следует выполнить команду

make r2

В данном случае ***r2*** является именем целевого файла, для создания которого будет выполнена команда, определяемая строкой 6.11 и совпадающая с командой 5.4. То, что файл ***r2*** реально создан не будет (если только его создание не предусмотрено в программе ***prog.px***), позволяет многократно выполнять действия 5.1 - 5.4 с помощью команды ***make r2***.

Основное преимущество данного подхода в том, что он гарантирует, что в случае модификации программы, соответствующая ее часть будет скомпилирована и выполняемый файл будет обновлен **до** того, как программа будет в очередной раз запущена. Кроме того, поскольку обновляться будут только те файлы, "источники" которых были действительно модифицированы **после** последнего запуска утилиты ***make***, этот подход позволяет экономить значительное время, особенно при разработке больших проектов, включающих десятки и сотни исходных файлов.

Тема 7. Проблема балансировки загрузки

Одной из основных и наиболее трудно решаемых задач разработки параллельных алгоритмов является проблема балансировки загрузки. С одной стороны, увеличение числа процессоров, между которыми распределена работа, уменьшает общее время решения многих задач (до некоторого числа процессоров). С другой стороны, распределение работы между процессорами приводит к необходимости передачи между ними данных, что замедляет вычисления.

При фиксированном числе процессоров работу между ними можно распределить по-разному. В связи с этим встает задача определения такого распределения работы для заданной задачи и заданного числа процессоров, при котором

- все процессоры выполняют примерно одинаковый объем работы - загружены равномерно;
- объем передаваемых данных минимизирован.

Несложно убедиться, что для многих задач эти требования являются противоречивыми. В общем случае, параллельную программу можно представить в виде некоторого взвешенного графа, вершины которого представляют собой элементарные задания, каждое из которых может быть выполнено любым процессором, а ребра - наличие связей между заданиями. При этом, вес каждой вершины следует выбрать пропорциональным времени выполнения соответствующего задания, а веса ребер - пропорциональными объему передаваемых между заданиями данных. Таким образом, проблема балансировки загрузки сводится к задаче разрезания графа на заданное число частей так, чтобы суммарный вес разрезанных ребер был минимальным, а суммарные веса вершин, входящих в разрезанные части, примерно равны.

Не всегда следует минимизировать общий вес разрезанных ребер. Например, при передаче большого количества коротких сообщений следует минимизировать число разрезанных ребер, поскольку основное время при передаче короткого сообщения тратится на инициализацию самого процесса передачи данных. Другой возможностью является минимизация максимального трафика между парой процессоров. Также не всегда следует стремиться к равномерному распределению работ. Во многих случаях полезнее минимизировать объем работы, пришедшейся на самый загруженный процессор.

И, наконец, полезно рассматривать передачу данных как один из видов работы, суммируя общий объем выполняемой процессором работы с общим объемом передаваемых/принимаемых им данных, умноженным на подходящий коэффициент. При таком подходе остается равномерно распределить получившуюся "совокупную" работу.

Сформулированная проблема попадает в класс так называемых NP-сложных[1] задач. В связи с этим, точное решение задачи для графов большого объема в настоящее время найти не представляется возможным. Однако для некоторых частных случаев решение хорошо известно - например, при моделировании задач газовой динамики с помощью явных разностных схем на четырехугольных регулярных сетках достаточно разбить сетку на полосы или клетки общим числом, рав-

[1] NP-сложные задачи - задачи, не разрешимые в настоящее время иначе, нежели с помощью полного перебора. Классической NP-полной задачей является задача установления изоморфизма (эквивалентности) двух произвольных графов. Для многих задач, в том числе для сформулированной, доказана их эквивалентность задаче "гомоморфизма графов". Для NP-сложных задач не известны алгоритмы, время выполнения которых растет как полином от объема задачи (размера соответствующего графа) или медленнее, что и дало название всему классу задач - Nonpolynomial (не полиномиальные).

ным числу процессоров, в соответствии с принципом геометрического параллелизма. В общем же случае задача балансировки загрузки решается приближенно.

Виды балансировки загрузки

Различают статическую и динамическую балансировки загрузки. Статическая балансировка выполняется перед началом вычислений. При этом делаются определенные предположения относительно времени выполнения частей программы (например, при решении уравнений газовой динамики с помощью явных разностных схем предполагается, что время обработки одинаково для всех узлов разностной сетки) и объема передаваемых между процессами данных.

Динамическая балансировка загрузки выполняется, когда время выполнения различных частей программы или эффективная производительность процессорных узлов изменяется в ходе решения задачи. Простейшим алгоритмом динамической балансировки загрузки является уже рассмотренный метод "коллективного решения".

Эффективная производительность процессорных узлов может изменяться, например, при выполнении параллельной программы на сети рабочих станций. При запуске на какой-либо станции посторонней программы любого пользователя эффективная производительность соответствующего вычислительного узла уменьшается.

Время выполнения различных частей программы изменяется от шага к шагу, например при моделировании с помощью разностных схем задач горения. Время обработки точек, в которых интенсивно протекают химические реакции (эти точки расположены на фронте пламени), значительно превышает время обработки точек, в которых горения не происходит (рис. 49). Одним из способов совместного численного решения задач газовой динамики и химической кинетики является расщепление по физическим процессам. На каждом шаге по времени можно сначала выполнить решение уравнений газовой динамики в предположении, что химические реакции не протекают, а затем уравнений химической кинетики в предположении, что не происходит взаимодействия между узлами расчетной сетки. При таком подходе газодинамические уравнения удобно и эффективно решаются согласно методу геометрического параллелизма, но тогда узлы сетки, которые захвачены фронтом горения, могут оказаться локализованными на небольшом числе процессоров. В связи с этим, для эффективного счета необходимо перераспределять "горячие" точки между всеми процессорами. Поскольку фронт пламени может смещаться в ходе решения задачи, статически это сделать невозможно, тогда как динамическое перераспределение точек приносит хорошие результаты. На рис. 49 приведена зависимость времени обработки точек расчетной сетки при горении в атмосфере метанового факела. Струя метана поступает под высоким давлением вертикально вверх из левого нижнего угла рисунка. Вертикальными линиями отмечены границы между областями, обрабатываемыми шестью процессорами. Хорошо видно, что основная работа по расчету фронта горения сосредоточена на процессоре № 3 - 65%, затем на процессоре №2 - 30% и только 5% на процессоре № 1.

Остальные процессоры не будут выполнять никакой полезной работы на этапе решения уравнений химической кинетики, если не провести динамическое перераспределение "горячих" точек.

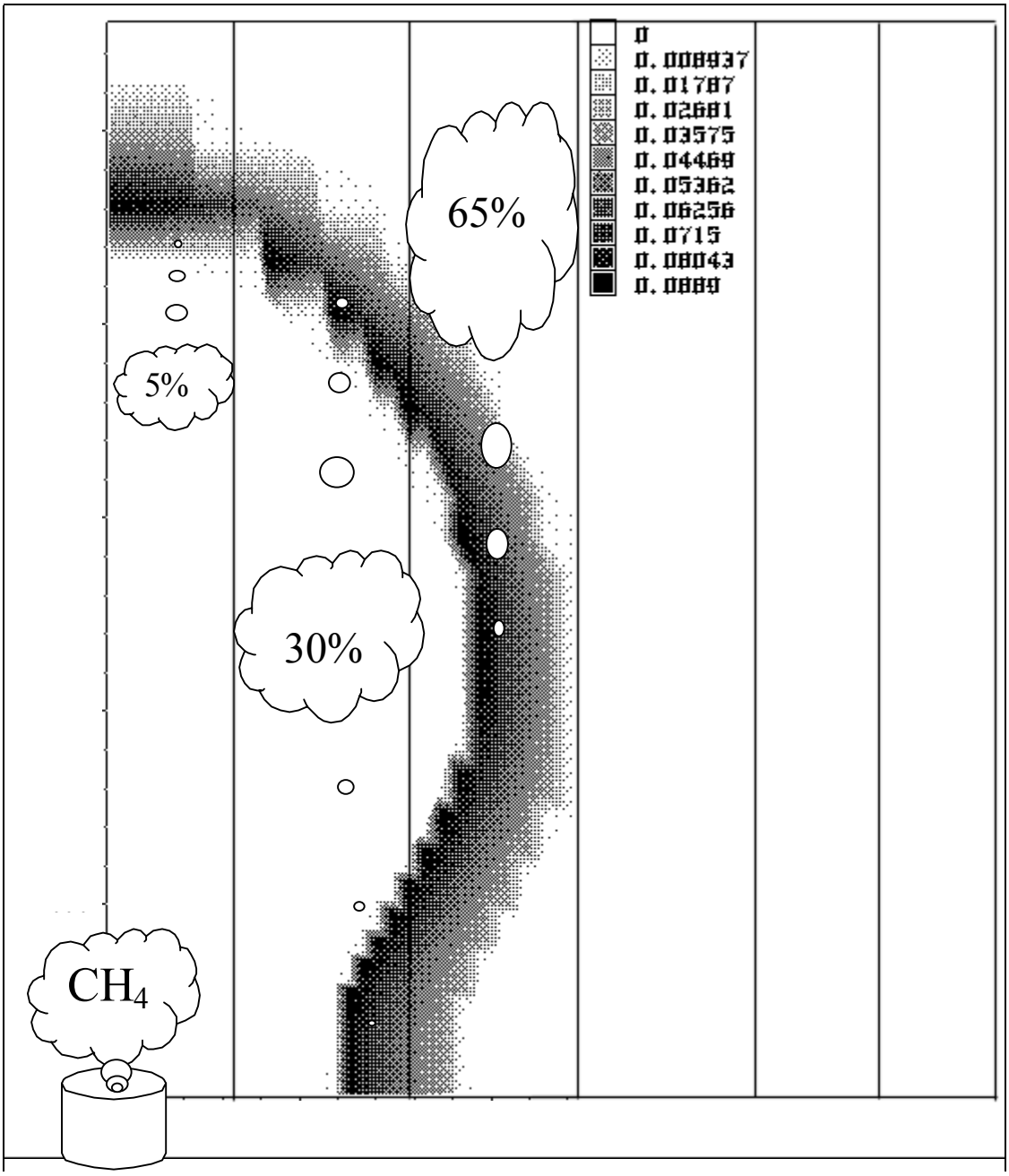

Рис. 49. Время (с), требующееся для расчета уравнений химической кинетики в узлах расчетной сетки на одном временном шаге

Разбиение нерегулярных сеток

Особую актуальность в настоящее время представляет проблема балансировки загрузки при решении задач на нерегулярных сетках (рис. 50), непосредственно сводимая к задаче разбиения графа на за-

данное число частей. Разбиение нерегулярной сетки на большое число частей по геометрическому принципу - разрезанием графа, изображенного на плоскости или в пространстве, дает в большинстве случаев плохие результаты. Несложно разбить вершины графа на примерно равные группы, но сложно обеспечить их "компактность" - сделать так, чтобы вершины каждой группы являлись соседними между собой, что уменьшает общее число ребер между группами.

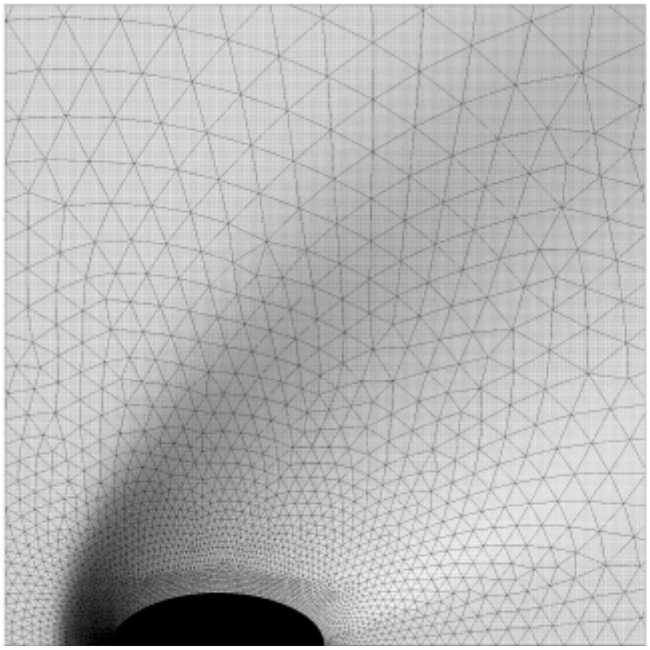

Рис. 50. Фрагмент нерегулярной сетки

Для графов небольшого объема существуют достаточно хорошие алгоритмы выполнения такого разбиения. Один из них - спектральный - будет рассмотрен ниже.

Для разбиения графов большого размера хорошо зарекомендовали себя иерархические алгоритмы, согласно которым разбиение вершин графа на требуемые группы осуществляется в несколько этапов:

1 - рекурсивное огрубление графа - уменьшение его размера,
2 - рекурсивная бисекция[2] вершин графа на заданное число групп,
3 - рекурсивное разукрупнение графа и локальное уточнение разбиения промежуточных графов.

Рекурсивное огрубление графа

Для уменьшения размера графа (уменьшения числа входящих в него вершин) удобно использовать рассматриваемый ниже алгоритм его огрубления. Полученный в результате первого огрубления умень-

[2] Бисекция графа - разбиение графа на две части.

шенный граф можно, в свою очередь, снова уменьшить. Построенная система вложенных графов может заканчиваться графом любого, сколь угодно малого размера.

Алгоритм огрубления графа может выглядеть следующим образом:

1 - выбрать некоторое множество ребер графа таким образом, чтобы никакие два ребра не имели общих вершин (рис. 51);

2 - "стянуть" пары вершин, соединенные выбранными ребрами, заменив каждую пару вершин одной. Соседями полученной вершины должны быть все соседи двух стягиваемых вершин, ее вес - сумме весов исходных вершин, веса ребер - сумме весов объединяемых ребер;

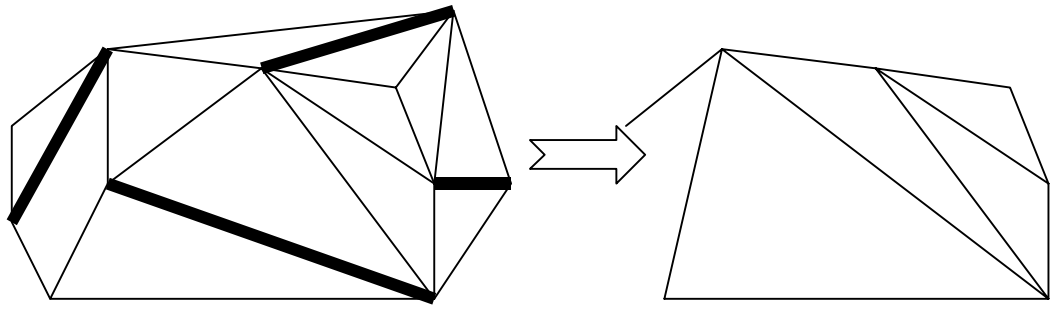

Рис. 51. Огрубление графа из 11 вершин до графа из 7 вершин

Рекурсивная бисекция

Для разбиения графа размером *n* (содержащего *n* вершин) на произвольное число частей *k* можно использовать рекурсивную процедуру разбиения графа на две части. Ее формальное описание выглядит следующим образом:

*7.1. bisect(**graf**(n),k1,k2)*
7.2. {
7.3. $n_1 = n\dfrac{k_1 + (k_1 + k_2 - 1)}{k_1 + k_2}$

$n_2 = n - n_1$

*7.4. if(k_1>1) bisect(**graf**(n_1),* $\dfrac{k_1+1}{2}, \dfrac{k_1}{2}$ *)*
7.5. else сформирована часть графа размером k_1

*7.6. if(k_2>1) bisect(**graf**(n_2),* $\dfrac{k_2+1}{2}, \dfrac{k_2}{2}$ *)*
7.7. else сформирована часть графа размером k2
7.8. }

Листинг 7. Алгоритм рекурсивной бисекции

Обратившись к этой процедуре с помощью вызова *bisect(graf(n),* $\frac{k+1}{2}$*,* $\frac{k}{2}$*)* мы получим требуемое разбиение. Нетрудно видеть, что если *k* четно ($k = k_1+k_1$), граф разбивается на очередном шаге на две равные части, в противном случае ($k=(k_1+1)+k_1$) граф разбивается на две части в пропорции $\frac{k_1+1}{k_1}$. Пример разбиения графа из 100 вершин на 7 частей примерно одинакового размера приведен на рис. 52.

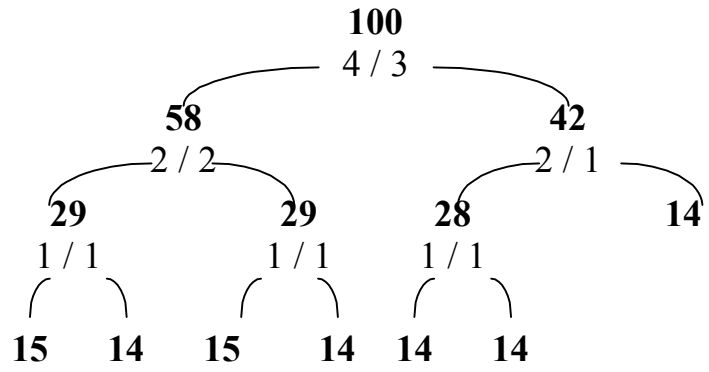

Рис. 52. Бинарное дерево разрезов

Спектральный алгоритм

Спектральный алгоритм эффективен для разбиения небольших графов на две части таким образом, чтобы число вершин, входящих в группы, были примерно равны, а вес разрезанных ребер был минимален. Алгоритм может быть адаптирован на случай, когда вершины графа имеют разные веса. Многочисленные эксперименты показали, что получаемое в результате его применения разбиение не хуже, а как правило, лучше разбиений, получаемых другими методами.

Рассмотрим (без доказательства) основную идею метода.

Пусть дан граф $G=(N,E)$, где N - множество вершин графа, E - множество его ребер, e_{ik} - веса ребер. Поставим в соответствие графу G, не содержащему петель и кратных ребер, матрицу $A(G)=(a_{ik})$, называемую Лапласианом графа G, такую, что

$$a_{ik} = a_{ki} = \ 0, \quad i \neq k \quad (i,k) \notin E;$$
$$a_{ik} = a_{ki} = -e_{ki}, \quad i \neq k \quad (i,k) \in E;$$
$$a_{ii} = -\sum_{k \neq i} a_{ik}, \quad i,k \in N.$$

Вычислим *a(G)* - второй наименьший собственный вектор *A(G)*. Для получения искомого разрезания графа следует упорядочить вершины графа *G* в соответствии с координатами вектора *a(G)* и разделить их на две группы, с большими и с меньшими значениями координаты, соответственно.

Наиболее трудоемким этапом при применении этого метода является поиск собственных векторов матрицы *A(G)*, что ограничивает размер разрезаемого графа.

Этап локального уточнения

После того, как огрубленный граф небольшого размера разбит на заданное число частей, следует восстановить исходный граф. Для этого следует каждую из полученных на этапе огрубления вершин разбить на две исходные и определить, в какую часть разрезанного графа они будут входить. Естественным представляется поместить полученные при разбиении вершины в ту же группу, в которую входила разбиваемая вершина. Для улучшения качества разбиения следует после очередного шага восстановления графа выполнить процедуру локального уточнения положения полученных вершин. Основная цель локального уточнения заключается в том, чтобы определить для каждой вершины ту группу, при перемещении в которую число разрезанных ребер будет минимально при сохранении равномерного распределения вершин по группам. Поскольку рассматриваются только единичные переносы вершин, общее их число пропорционально произведению числа групп и среднего числа вершин в группе, что позволяет решить задачу за ограниченное время при следующих соглашениях:

- при каждом уточняющем шаге каждая вершина переносится из одного набора в другой не более одного раза;
- на каждом цикле выполняется не более некоторого заданного числа, например при оптимизации графов с числом вершин < 5000 можно ограничиться 500 перемещениями.

Примеры разбиения нерегулярного графа, содержащего 75790 точек и 226300 ребер на 32 части в соответствии с исходной нумерацией вершин и в соответствии с приведенным иерархическим алгоритмом, приведены на рис. 53, 54, а данные о разрезанных ребрах - в таблице 5.

Таблица 5

Разбиение графа

Число процессоров	Число разрезанных ребер		Во сколько раз уменьшилось
	при начальном разбиении	при спектральном разбиении	
8	12406	2664	4.7
12	16553	3109	5.3
32	34612	5474	6.3

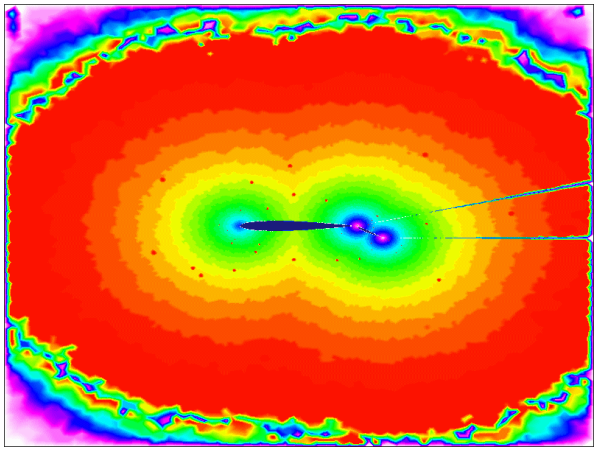

Рис. 53. Разбиение на 32 части, в соответствии с исходной нумерацией вершин

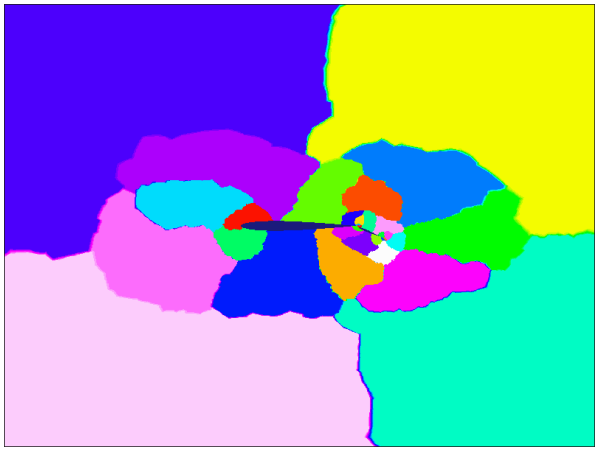

Рис. 54. Спектральное разбиение на 32 части

Тема 8. Визуализация в распределенных вычислительных системах

Быстрое развитие многопроцессорной вычислительной техники привело к возникновению ряда проблем, нетипичных для вычислительных систем традиционной архитектуры. Речь идет о факторах, не связанных непосредственно с проблематикой решаемых на суперкомпьютерах задач и сложностью создания соответствующих параллельных алгоритмов. Тем не менее обсуждаемые ниже аспекты значительно затрудняют использование систем массового параллелизма, тем самым сдерживая их дальнейшее развитие.

Говоря о проблемах разработки математического обеспечения и использования многопроцессорных систем, на первом месте традиционно отмечают сложность адаптации последовательных алгоритмов на параллельные системы вообще и на системы с распределенной памятью в особенности. Хорошо известно, что множество традиционных алгоритмов не имеют эффективных параллельных аналогов. В связи с этим основные силы на протяжении длительного времени сконцентрированы на поиске и изучении новых подходов к построению параллельных алгоритмов. Здесь и алгоритмы, специфичные для конкретных задач, и целые классы принципиально новых алгоритмов, ярким представителем которых являются нейросетевые алгоритмы. Эти усилия обеспечили несомненный прогресс в области разработки параллельных алгоритмов для решения прикладных задач широкого круга.

В качестве следующего препятствия на пути эффективного использования параллельной техники справедливо называют проблемы обеспечения сбалансированной загрузки процессоров в ходе решения задачи. В этом направлении также ведутся интенсивные исследо-

вания, известен широкий спектр алгоритмов как статической, так и динамической балансировки загрузки. Однако, несмотря на ощутимые успехи в решении отмеченных задач, типичный коэффициент интегральной загрузки большинства систем массового параллелизма составляет по разным оценкам всего от 10% до 3%. И это несмотря на наличие алгоритмов и программ, обеспечивающих весьма высокий (часто более 90%) коэффициент распараллеливания при решении многих задач.

При переходе на технику массового параллелизма значительно возрастает нагрузка на разработчика прикладного обеспечения, что плохо уже само по себе, но сплошь и рядом ситуация усугубляется стрессом, связанным с отсутствием в новых системах привычных инструментов разработки программ и анализа получаемых результатов. Отметим, что в такую же стрессовую ситуацию попадает и системный администратор многопроцессорного комплекса. Первое приводит к значительным затратам на адаптацию уже не алгоритмов, но коллектива разработчиков к новой ситуации - процесс, занимающий не один месяц и даже не один год; второе приводит к многочисленным простоям вычислительной системы, даже в условиях наличия достаточного набора задач, способных в принципе обеспечить близкую к 100% загрузку системы - служба системного администратора просто не в состоянии обеспечить эффективную дисциплину запуска задач без соответствующего системного обеспечения, а оно отсутствует практически всегда. Вероятно, пройдет заметное время, прежде чем многопроцессорные системы станут стандартом де-факто и появятся надежные распределенные операционные системы с адекватными и удобными средствами управления.

Значительной проблемой следует признать вопросы хранения, обработки и визуализации больших объемов данных, неизбежно сопровождающих проводимые масштабные вычислительные эксперименты. Так или иначе, визуализацией данных занимаются практически во всех исследовательских центрах, обладающих техникой нетрадиционный архитектуры.

Рассмотрим в качестве примера задачу визуализации результатов численного моделирования обтекания летательных аппаратов, движущихся со сверхзвуковыми скоростями, а именно обтекание небольшой выемки в корпусе летательного аппарата. Ее актуальность обусловлена тем, что многие приборы и устройства, которыми оснащены самолеты и ракеты, заглублены в корпус аппарата. Простой моделью может служить двумерная задача о сверхзвуковом обтекании прямоугольной двумерной полости вязким потоком (рис. 57). Данная проблема требует вычислений с высокой точностью, для чего необходимо использование подробных сеток по времени и пространству и, как следствие, высоких вычислительных затрат.

Качество визуализации играет определяющую роль при изучении и интерпретации численных результатов. Использование при моделировании подробных сеток приводит к тому, что файлы, содержащие результаты вычислений, достигают значительных размеров (уже при использовании всего лишь 6 процессоров их размер составляет десятки мегабайт). Определенно, точность вычислений (а следовательно, и размер файлов) ограничивается только скоростью, с которой вычислительная система может выполнять расчет задачи. Скорость вычислительных систем в настоящее время быстро растет, в многопроцессорных системах увеличивается число процессоров, и поэтому в самом скором времени стоит ожидать значительного увеличения объема файлов с результатами вычислений. Это предъявляет очень жесткие требования к системам визуализации данных и приводит к появлению новых методов обработки информации.

Сложность визуализации данных большого объема определяется рядом факторов, основные из которых следующие:

- оперативной памяти и вычислительной мощности последовательных систем недостаточно для обработки результатов моделирования;
- разрешение устройств вывода данных уже не соответствует объему данных - при использовании неравномерных расчетных сеток сотни узлов сетки могут попадать в одну точку (*pixel*) экрана, полностью лишая исследователя возможности увидеть что-либо в областях сгущения сетки;
- наборы данных, порождаемые при решении задач, описываемых большим числом пространственных функций (например задач химической кинетики, описывающих взаимодействие десятков химических веществ), требуют принятия специальных мер по организации их хранения - использование для каждого набора данных отдельного файла значительно затрудняет функционирование файловой системы (из-за очень большого числа файлов) и, несмотря на отсутствие принципиальных ограничений, делает работу с ними практически невозможной;
- проблема снижения трафика (объема данных, передаваемых между частями системы) при визуализации с удаленных рабочих мест занимает не последнее место в этом списке, поскольку практически вся работа с многопроцессорными системами осуществляется через локальную и глобальную сети.

Среди основных требований к системе визуализации отметим интерактивность, возможность изучения произвольного фрагмента моделируемого скалярного или векторного поля, возможность просмотра последовательности файлов в режиме мультипликации.

Технология клиент/сервер

Современные требования к программам, интерактивно взаимодействующим с пользователем, не допускают долгой реакции на действия пользователя. Практика показывает, что для обеспечения высокой скорости работы визуализатора все необходимые данные желательно хранить в оперативной памяти, что практически невыполнимо для программы, работающей на обычном персональном компьютере. В связи с этим разумно использовать параллельные системы не только для численного моделирования задачи, но и для визуализации данных. Естественным будет построение системы визуализации данных на основе перспективной и популярной в последнее время технологии клиент/сервер (рис. 55). Сервер запускается на мощной многопроцессорной системе и обслуживает многих клиентов, работающих на своих персональных компьютерах. При этом сервер производит все расчеты и пересылает подготовленные данные клиентским частям, которые занимаются только отображением полученных данных и передачей запросов на обработку данных, полученных от пользователя. Поскольку передавать надо только те данные, которые непосредственно описывают формируемое на экране изображение, их объем может быть сравнительно небольшим. Как будет показано ниже (рис. 56), этот объем может практически не зависеть от размера визуализируемого файла и величины изучаемого фрагмента.

Мощные вычислительные системы работают, как правило, под управлением операционной системы, совместимой с Unix, графическая подсистема которой имеет возможность открывать окна на дисплее другого компьютера, в том числе и персонального, работающего под управлением одной из популярных пользовательских оболочек или операционных систем. Казалось бы, зачем создавать сложную систему на основе технологии клиент/сервер, если можно запустить визуализатор на многопроцессорной системе, назначив в качестве устройства вывода графической информации дисплей удаленного компьютера пользователя? Но эффективность такого решения невелика, так как пропускная способность сети ограничена, потребности в ней очень высоки, а при таком подходе по сети придется передавать большой объем графической информации. Как правило, этого можно избежать. Например, зачем передавать изображения векторов, когда можно передать только их положение и координаты (объем передаваемых по сети данных уменьшается при этом на несколько порядков). Современные компьютерные сети, особенно глобальные, не настолько быстры, чтобы обеспечить передачу большого объема данных за приемлемое время, так что сокращение объемов передаваемой информации очень важно.

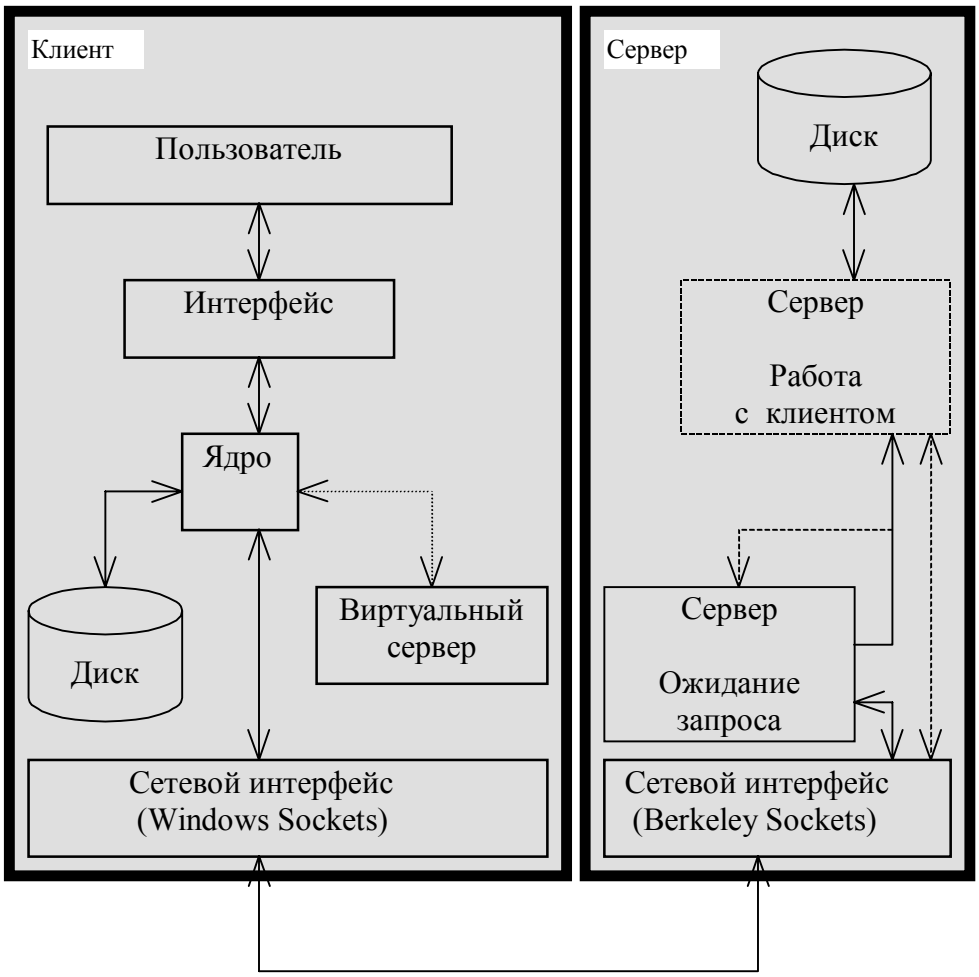

Рис. 55. Структура системы визуализации

Подход на основе технологии клиент/сервер имеет еще много важных преимуществ. В частности, переносимость. Парк вычислительных систем обновляется очень быстро, все время появляются новые, более мощные вычислительные системы. При переходе на новую вычислительную систему приходится переносить на нее разработанное программное обеспечение. Облегчить этот процесс призваны стандарты на программный интерфейс и возможности операционной системы. Но всегда существуют накладки и несоответствия, особенно это касается только недавно введенных стандартов. На многие возможности системы стандартов не существует вообще. В частности, вполне может оказаться, что на новой системе нет привычной графической подсистемы. В то время как поддержка математики и сетевые функции издавна подчиняются очень жестким стандартам, и потому программы, использующие только математику и сетевые возможности, переносить очень легко - зачастую достаточно просто заново скомпилировать программу на новой системе. При этом вся графическая часть локализована на клиентской машине и может работать без изменений с новым сервером, работающим на новой системе.

Вторым важным преимуществом технологии клиент/сервер является универсальность. В последнее время для персональных компьютеров появилось много популярных операционных систем и оболочек - например, OS/2, Windows, Windows ‘95, Windows NT, Unix. Каждая из них обладает своими особенностями, важными для разных категорий пользователей, и поэтому для персональных компьютеров нет единого стандарта. Приложения, созданные для одной из этих операционных систем, либо вообще не работают под управлением других операционных систем, либо работают, но с ограничениями. Под каждую из этих операционных систем можно написать свою клиентскую часть, которая будет использовать возможности конкретной операционной системы, в то время как сложная математика останется на сервере и не подвергнется изменениям, которые могут внести трудноуловимые ошибки. При этом каждый пользователь сможет анализировать результаты расчетов в привычной и удобной для него операционной системе.

Третьим преимуществом технологии клиент/сервер является возможность обеспечения удаленного доступа. В последнее время большой популярностью пользуются не только локальные компьютерные сети, но и глобальные. Ярким и широко известным представителем такой сети является Internet. Пользователи имеют возможность, используя ресурсы сети, обращаться к удаленным суперкомпьютерным системам. Как уже было отмечено, файлы с данными моделирования могут иметь очень большой размер, транспортировать их на физических носителях или передавать по сети в их исходном виде очень неудобно и требует большого количества ресурсов. При проведении вычислительных экспериментов на удаленном компьютере все данные хранятся на нем, а исследователь сможет с помощью технологии клиент/сервер работать непосредственно с визуальным представлением результатов своих расчетов, используя клиентскую часть визуализатора и ресурсы глобальной сети.

Подчеркнем, что использование многопроцессорных систем при визуализации больших объемов данных в последнюю очередь преследует своей целью уменьшение времени построения изображения. Если данные *могут* быть размещены в оперативной памяти одного процессора, последовательная программа, в большинстве случаев, справится с построением таких изображений, как изолинии, изоповерхности, линии тока и т.д. быстрее, чем многопроцессорная, поскольку затраты на распределение исходных данных между процессорами оказываются сопоставимы, или даже превышают затраты на формирование изображения. Но если объем данных *превышает* объем оперативной памяти одного процессора, тогда использование многопроцессорной системы вполне оправдано и, во многих случаях, неизбежно.

Снижение объема передаваемых данных

Как отмечалось, пользователь может выбрать на экране интересующую его область, для того чтобы рассмотреть ее подробнее - "растянуть" на всю поверхность текущего окна, увеличив масштаб изображения. При этом возникает следующая проблема: исходная сетка может содержать очень большое число узлов, а пользователь хочет получить общее представление о характере поля или, наоборот, пользователь рассматривает участок векторного или скалярного поля с очень большим увеличением, и сетка в этом месте содержит мало узлов. Особенно остро проблема проявляется при визуализации векторных полей с помощью "стрелочных" изображений. Получается, что в первом случае узлов слишком много для того, чтобы их можно было корректно изобразить на экране: если попытаться нарисовать все векторы, то их изображения заполнят собой значительную часть окна, они будут накладываться друг на друга или будут настолько мелкими, что будут неинформативны, не отличаясь от точек. Во втором случае на экране получится один или два вектора. В обоих случаях получить адекватное представление о характере распределения скоростей в поле не удастся. После многих экспериментов было найдено следующее решение - рисовать всегда одинаковое количество векторов вне зависимости от масштаба и шага сетки в рассматриваемой области (рис. 56). Это дает приятный эффект будто бы непрерывного, а не дискретного задания векторного поля. Изображаемые векторы получаются путем интерполяции внутри клетки, в которой лежит точка, где нужно изобразить вектор.

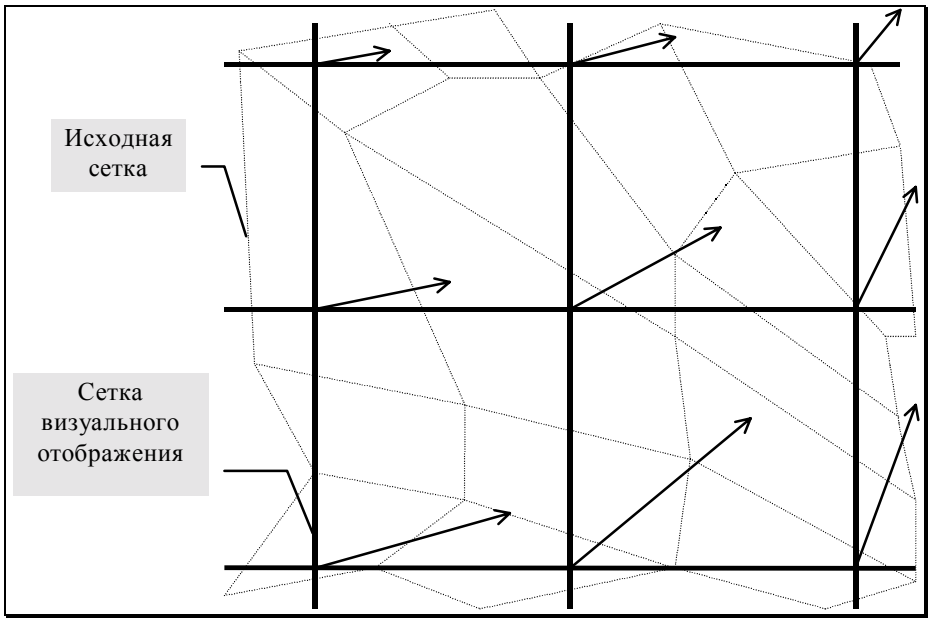

Рис. 56. Схема генерации сетки визуального отображения

Поскольку высокая точность при визуализации не требуется, вполне достаточно использовать для расчета векторов сетки визуаль-

ного отображения билинейной интерполяции, требующей минимального числа вычислительных операций. В общем случае произвольной выпуклой четырехугольной сетки метод нахождения нужной клетки требует больших вычислительных затрат и перебора клеток сетки. Для снижения этих затрат можно использовать хеширование[3] по геометрическому признаку.

Сжатие сеточных функций

Рассматривая проблему сжатия функций результатов численного моделирования будем предполагать, что данные представлены вещественными функциями, заданными на регулярных четырехугольных сетках. Преобразование к формату, пригодному для вывода непосредственно на дисплей, приводит к значительному огрублению данных. Например, использование 256-цветной палитры для отображения скалярных функций, фактически эквивалентно тому, что при отображении используется только 256 градаций из значений. Между тем, вещественные числа одинарной точности (4 байта) позволяют оперировать 2^{24} градациями. Изображение, построенное с применением большего числа разных цветов, безусловно выглядит привлекательнее, но объективно не несет, при восприятии глазом человека, намного больше информации, чем 256-цветное, по крайней мере, если речь идет о изображении изолиний или векторов. Таким образом, для хранения результатов расчетов вполне можно использовать вещественные числа одинарной точности.

Основное препятствие при сжатии многомерных массивов вещественных чисел заключается в том, что даже близкие по величине вещественные числа имеют в двоичном эквиваленте значительно различающийся вид. Гладкие, слабо изменяющиеся по пространству сеточные функции плохо упаковываются стандартными алгоритмами, поскольку двоичное представление задающих их чисел выглядит весьма беспорядочно, причем последние байты (содержащие младшие знаки мантиссы) двоичного представления вещественных чисел выглядят практически случайными, тогда как первые байты чисел (содержащие значение характеристики и старшие знаки мантиссы) с близкими зна-

[3] Например, всю исходную область можно разбить на достаточно крупные прямоугольные клетки и подготовить для каждой клетки списки попадающих в нее четырехугольников. Причем, четырехугольник, имеющий с клеткой хотя бы одну общую точку, следует считать попадающим в эту клетку. Тогда для определения четырехугольника, в который попал узел сетки визуального отображения, достаточно сначала определить, в какую прямоугольную клетку он попал, а потом конкретный четырехугольник - полным перебором по четырехугольникам этой прямоугольной клетки. При удачном выборе размера прямоугольных клеток (он может быть и переменным) среднее число перебираемых четырехугольников можно сделать значительно меньшим, чем их общее число. Процедуры такого рода называют процедурами хеширования.

чениями похожи между собой. Сгруппировав перед сжатием функций байты таким образом, чтобы отдельно упаковывался массив первых байт, потом массив вторых байт, и так далее, можно с помощью простого алгоритма группового кодирования получить значительное сжатие первых двух - трех массивов. Массив последних байт как правило практически не удается сжать, однако оказывается, что информация, записанная в нем, практически не влияет на вид формируемых при визуализации изображений и может быть в большинстве случаев отброшена. Отбрасывая некоторое число последних бит мантиссы можно варьировать коэффициент сжатия функций, практически не теряя в качестве представления данных, что подтверждают рисунки 57-58, на которых приведены изображения, построенные на основе исходных и сжатых с потерями данных.

Таблица 6

Сжатие сеточных файлов

	Коэффициенты сжатия				
Сетка, число узлов	Размер несжатых данных, байт	Сжатие утилитой gzip	Сжатие без потерь	Сжатие с потерями 0.003%	Сжатие с потерями 0.78%
121x21	33 037	4.28	3.07	7.07	14.57
601x751	7 672 991	3.47	2.12	6.63	14.16
2 101x451	12 318 167	5.30	3.24	9.86	20.56

В таблице 6 приведены коэффициенты сжатия, полученные при упаковке данных, заданных на четырехугольных сетках с помощью стандартной утилиты архивации gzip и с помощью разработанных алгоритмов сжатия сеточных функций с потерями и без потерь.

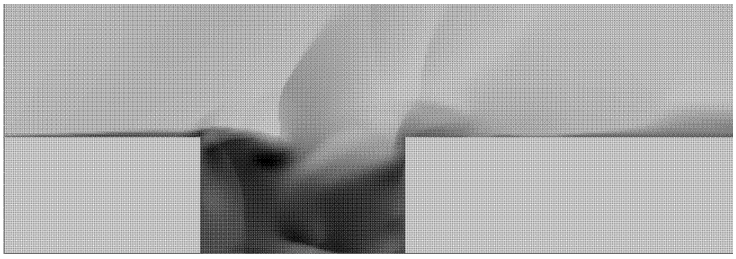

Рис. 57. Несжатые данные

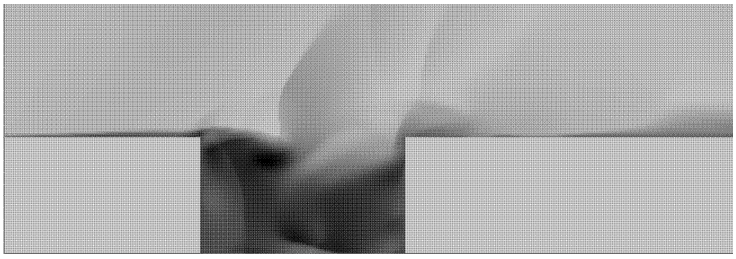

Рис. 58. Сжатые данные (коэффициент сжатия 20.56, потери 0.78%)

Библиографический список

Параллельные системы и процессы

1. *C A R Hoare.* Communicating sequential processes. Communications of the ACM, 21(8):666-677, Aug 1978.
2. *C A R Hoare.* Communicating sequential processes. In B. Shaw, editor, Digital Systems Design. Proceedings of the Joint IBM University of Newcastle upon Tyne Seminar, 6-9 September 1977, pp. 145-56. Newcastle University, 1978.
3. *C A R Hoare.* Transputer application. Manual. - Prentice Hall London, 1984.
4. http://www.comlab.ox.ac.uk/oucl/people/cvbib/hoare.html
5. *Flynn M.J.* Some Computer Organizations and their Effectiveness. // IEEE Trans Comput., vol. C-21, pp. 948-960, 1972.
6. *Dijkstra E.W.* Solution of a Problem in Concurrent Program Control. // CACM, Vol. 8, No. 9, Sept. 1965, p. 569.
7. *Dijkstra E.W.* Cooperating Sequential Processes in Programming Languages, ed. F.Genuys, Academic Press, New York. - NY, 1968.
8. Языки программирования. Под ред. Ф.Женуи, Пер. с англ. В.П.Кузнецова, Под ред. В.М.Курочкина. - М.: Мир, 1972, 406 стр.

Визуализация

9. *Джамса К., Коуп К.* Программирование для Internet в среде Windows. - Санкт-Петербург: Питер пресс, 1996.
10. *Шенен П., Коснар М., Гардан И., Робер Ф., Робер И., Витомски П., Кастельжо П.* Математика и САПР. - М.: Мир, 1988.
11. PARIX 1.3 for Power PC: Software documentation and Reference manual, Parsytec Computer GmbH, 1994.
12. SunOS 5.2 Network interfaces programmer's guide, Sun Microsystems, 1992.
13. Визуализация гидродинамических потоков. Отчет ИОС РАН.
14. *Cebral J. R.* ZFEM: Collaborative visualisation for parallel multidisciplinary applications. Parallel CFD '97: Recent development and advances using parallel computes, Manchester, May 19-21, 1997, Preprints.
15. Microsoft developer network: Microsoft development library, April '95.

Балансировка загрузки

16. *Hendrickson B. and Leland R.* A Multi-Level Algorithm for Partitioning Graphs, Tech. Rep. SAND93-1301, Sandia National Laboratories, Albuquerce, October 1993.

17. *Hendrickson B. and Leland R.* An Improved Spectral Graph Partitioning Algorithm For Mapping Parallel Computations — SIAM J. Sci. Comput., 1995, vol.16. № 2.
18. *Fiedler M.* Eigenvectors of aciyclic matrices. - Praha, Czechoslovak Mathematical Journal, 25(100) 1975, pp. 607-618.
19. *Fiedler M.* A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. - Praha, Czechoslovak Mathematical Journal, 25(100) 1975, pp. 619-633.
20. *Хейгеман Л., Янг Д.* Прикладные итерационные методы. - М.: Мир, 1986. - 448 с.
21. *Kershaw D.S.* The incomplete cholesky-conjugate gradient method for the iterative solution of linear equations. // J. Comput. Phys. - 1978. - Vol. 26, N 1. - p. 43-65.
22. *Bruce Hendrickson and Robert Leland.* An Improved Spectral Partitioning Algorithm for Mapping Parallel Computations. // SIAM J. Comput. Phys. - March 1995. - Vol. 16, N 2. - p. 452-469.
23. *B.N. Parlett, H.Simon and L.M.Stringer.* On Estimating the Largest Eigenvalue with the Lanczos Algorithm. // Mathematics of computation - March 1995. - Vol. 38, Number 157. - p. 153-165.
24. *Fiduccia C.M. and Mattheyses R.M.* A Linear-Time Heuristic for Improving Network Partitions. - 19th Design Automatic Conference 1982. - paper 13.1 - p. 175-181.
25. *Simon H.D.* Partitioning Of Unstructured Problems For Parallel Processing — Computing Systems in Engineering, 1991, vol.2, № 2/3.
26. *Евстигнеев В.А.* Применение теории графов в программировании./ Под ред. А.П.Ершова. - М.: Наука, Главная редакция физико-математической литературы, 1985-352 с.

Разное

27. *Дж. Ортега.* Введение в параллельные и векторные методы решения линейных систем: Пер.с англ-М.:Мир, 1991, с.24, с.34.
28. *Аляутдинов Д.А., Далевич А.Н.* Параллельный СИ (PARALLEL C). М.: МАИ, 1991, 112 с.

29. *Коновалов А.Н.* Введение в вычислительные методы линейной алгебры. Новосибирск: ВО ”Наука”. Сибирская издательская фирма, 1993, 159 с.
30. Транспьютеры. Архитектура и программное обеспечение. Пер. с англ. / Под ред. Г. Харпа. - М.: Радио и связь, 1993, 304 с., ил.
31. Высокоскоростные вычисления. Архитектура, производительность, прикладные алгоритмы и программы суперЭВМ: Пер. с англ./ Под ред. Я.Ковалика. - Москва: Радио и связь, 1988, 432 с.
32. *Р.Бэбб, Дж.Мак-Гроу, Т.Акселрод* и др. Программирование на параллельных вычислительных системах: Пер. с англ. /под ред. Р.Бэбба II. - М.:Мир, 1991, -376 с., ил.
33. Архитектура ЭВМ и численные методы. Сб. науч. трудов / Под ред. В.В.Воеводина. М.: ОВМ АН СССР, 1983г. - 142 с.
34. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. М.: Пер. с англ. - Мир, 1981, 368 с.
35. *Митчел Д.А.П., Томпсон Дж.А., Мансон Г.А., Брукс Г.Р.* Внутри транспьютера. - М.: Мейкер, 1992, 206 с.
36. Оре О. Теория графов. Москва: Наука,1980, 336 с., перевод с англ.
37. *Маркин М.А., Лопатин В.А.* Т9000. // Журнал д-ра Добба N 3, 1991, стр. 34-37.
38. *Кульков Г.Б., Ялин В.В.* Сравнительный анализ временных характеристик модулей ТТМ100, ТТМ110 и ВМ106. // В сборнике тезисов 3-й конференции Российской Транспьютерной Ассоциации. - Москва, 1993
39. *Бройтман Д.* Микропроцессор Power-601. // Монитор N 4, 1994, стр. 56-61.

Приложения

Алгоритм децентрализованного управления и распределенной обработки глобальной информации

Рассмотрим один из эффективных методов организации итераций при решении задач математической физики с помощью итерационных методов на примере решения уравнения Лапласа, описывающего поведение давления на поле нефтяного месторождения. Как правило, для достижения заданной точности при решении систем алгебраических разностных уравнений требуется, в зависимости от внешних данных (в основном от мощности скважин), существенно разное число итераций. Это число сильно и плохо предсказуемо меняется от одного временного слоя к другому. Можно обойтись одной-двумя итерациями при установившемся течении в конце эксплуатации месторождения, однако в начале, когда резко меняется водонасыщенность пласта (доля воды в единице объема смеси воды и нефти), или в

моменты смены режимов на скважинах, например для воздействий, связанных с переводом для интенсификации процесса нефтедобычи добывающих скважин под нагнетание и наоборот, число итераций при решении уравнения Лапласа может возрасти до нескольких сотен. Кроме того, при распределенной обработке итерации на части процессоров могут сойтись, тогда как на других еще нет - возникает вопрос: в какой момент итерации остановить? В системах с общей памятью он имеет простой ответ, поскольку в любой момент времени вся необходимая информация доступна управляющей программе. В системе с распределенной памятью это уже не так. В такой системе не ясно, кто должен принимать решение об окончании итераций, на основе какой информации, как она будет ему доставлена и, наконец, как процессоры сети узнают, что такое решение принято.

Возможное решение состоит в пересылке на каждой итерации невязки (величины изменения приближенного решения на очередной итерации) с каждого процессора на управляющий, который, проанализировав состояние всех задач, возвращает им признак, надо ли продолжать итерации, или уже можно переходить к следующему шагу по времени. Решение привлекает своей простотой, однако при таком подходе значительную часть времени между итерациями процессоры будут простаивать. Можно проводить такую процедуру не на каждой итерации, а через заранее заданное фиксированное число итераций (например M). Если это число будет заметно меньшим общего числа итераций, потери будут велики, если оно будет приближаться к нему, то однажды собрав данные, в момент, когда итерации почти сошлись, мы заставим систему следующие M итераций уточнять уже полученное решение и потеряем время на таких “лишних” итерациях.

Рассмотрим децентрализованный алгоритм распространения глобальной информации и принятия решения. Под “глобальными” подразумеваются данные, которые желательно было бы разместить в общей памяти, доступной одновременно всем процессорам сети (например, шаг Куранта, имеющий разное локальное значение в областях моделируемых разными процессорами сети, невязка и т.д.). За неимением общей памяти ее обычно эмулируют сбором данных со всех процессоров на корневой, их последовательной обработкой и рассылкой результата обратно по всем задачам. Алгоритм децентрализованного принятия решения позволяет выполнить аналогичные функции на каждом процессоре, минуя этапы сбора на единственном корневом процессоре и их обратной рассылки.

Для этого каждый процессор сети должен уметь следующее:

- анализировать глобальную информацию;
- своевременно информировать сеть о собственном состоянии и о состоянии тех процессоров, о которых к нему поступила информация;

- самостоятельно принимать решение об окончании итераций.

При этом предлагается совершенно исключить этап обратной рассылки признака окончания итераций, а этап сбора невязок заменить на несколько "лишних" итераций, причем их число фиксировано и, как показано ниже, невелико.

Приведем описание децентрализованного алгоритма принятия решения об окончании итераций.

Рассмотрим сеть, состоящую из *N*M* (на рис. 28 *N=4, M=8*) процессоров, соединенных в решетку. Пусть каждый процессор на каждой итерации сообщает своим "соседям", сошлись у него и у тех процессоров, информация от которых к нему уже поступила, итерации или нет. Это можно делать с минимальными потерями времени, объединяя при передаче массив соответствующих признаков вместе с новыми значениями в граничных точках.

При таком подходе процессор P_{22} (рис. 28) узнает о том, что у процессора P_{00} итерации сошлись, только через *R* тактов (*R*=4) после того, как это в действительности произошло (будем для разнообразия использовать термин "такт" как синоним слова "итерация").

Предположим, что в P_{00} итерации сошлись. Тогда P_{01}, получив эту информацию на такте t^0, на такте t^1 должен передать ее процессорам P_{02} и P_{11} вместе с данными о себе, включив после соответствующего преобразования в *матрицу* **G** *признаков состояния всех процессоров* сети. В свою очередь, он получает от соседей такие же матрицы **G**, отражающие их представления о состоянии системы. Матрица **G** содержит *N*M* чисел от *-D* до *D+1*, в начальном состоянии она содержит только нулевые значения. Если в процессе расчета оказывается, что $G_{ij}=D+1$, то это означает, что итерации в P_{ij} уже сошлись и сведения об этом распространились по всем процессорам сети. Если $D \geq G_{ij} \geq 0$, то либо итерации в P_{ij} еще не сошлись, либо еще не все процессоры знают, что итерации в P_{ij} уже сошлись. Элемент матрицы G_{ij} первый раз принимает значение 1 в момент, когда в соответствующем ему транспьютере P_{ij} сойдутся итерации. Далее это значение на каждом такте передается соседним по графу транспьютерам, одновременно увеличиваясь на 1, что позволяет интерпретировать его, как значение времени, прошедшего с момента получения решения. Оно замечательно тем, что течет одинаково для **всех** транспьютеров, в которых оно отлично от нуля. Именно это свойство обеспечивает синхронность принятия решения об окончании итераций.

Возможен третий вариант: $-D \leq G_{ij} < 0$. Рассмотрим случай, когда в P_{00} только что включилась скважина, а вся остальная область содержит ровный фон, итерации только начались. Очевидно, что во всех P_{ij}, кроме содержащего источник, первая же итерация сойдется, но через несколько тактов, когда возмущение достигнет краев расчетной области P_{00}, условие окончания итераций может нарушиться и в P_{01}, и

в P_{10}, а потом и в остальных процессорах. Неизвестно, где в таком случае итерации сойдутся в последнюю очередь. Именно на этот случай и зарезервированы отрицательные значения в матрице **G**. Они используются, если процессор, заявивший ранее о том, что у него итерации сошлись, обнаруживает, что теперь они расходятся.

Теперь можно уточнить, как обрабатывать матрицы $\mathbf{G^k}$, полученные от соседей, здесь k=1, ..., m, где m - число процессоров, соединенных с $P_{i_0 j_0}$. Каждый процессор $P_{i_0 j_0}$ формирует свою матрицу $\mathbf{G^0}$ по следующему алгоритму:

1. от соседних транспьютеров P^k получить значения искомой функции в граничных точках и матрицы признаков $\mathbf{G^k}$, k=1, ..., m;
2. вычислить значения искомой функции на новой итерации и в соответствии с ними определить, сошлись итерации на текущем процессоре или нет;
3. для каждого элемента [i,j] матрицы $\mathbf{G^0}$:
   3.1. если $G^k_{ij} < 0$ или ($G^k_{ij} > 0$ и $G^0_{ij} \geq 0$), то $G^0_{ij} = G^k_{ij}$ (k=1...m);
   3.2. если $G^0_{ij} \neq 0$ и $G^0_{ij} < D+1$, то $G^0_{ij} = G^0_{ij} + 1$.

Следующие два шага алгоритма относятся к обработке элемента $[i_0,j_0]$ матрицы $\mathbf{G^0}$, соответствующего "текущему" процессору $P_{i_0 j_0}$, то есть обновляется информация о своем состоянии:

4. если $G^0_{i_0 j_0} = 0$, и итерации сошлись, то $G^0_{i_0 j_0} = 1$;
5. если $G^0_{i_0 j_0} > 0$, а итерации разошлись, и существует соседний с $P_{i_0 j_0}$ процессор $P_{i_k j_k}$, в котором итерации не сошлись, т.е. $G^k_{i_k j_k} < 0$, то $G^0_{i_0 j_0} = -D$.

Полученная матрица $\mathbf{G^0}$ пересылается соседям на следующем такте.

Если все элементы получившейся в результате матрицы $\mathbf{G^0}$ равны D+1, то можно утверждать, что итерации сошлись во всех процессорах, и все они об этом оповещены, а значит, можно переходить к следующему шагу по времени.

Информация о том, что итерации сошлись в каком-либо конкретном процессоре, станет известна всем остальным не долее, чем через D тактов. К этому моменту соответствующий разряд матрицы $\mathbf{G^0}$ станет равен D+1, так как при каждом такте ненулевые разряды увеличиваются на единицу. Аналогично информация о внезапно разошедшихся где-либо итерациях станет известна всем также не более, чем за D тактов. Через время D соответствующий разряд матрицы $\mathbf{G^0}$ станет равен 0, и система "забудет", что итерации когда-то сошлись, а потом вновь разошлись, то есть вернется в исходное состояние.

Условие 5 гарантирует, что в том маловероятном случае, когда слабое возмущение (само в пределах допустимой точности) на границе между транспьютерами приводит к тому, что итерации в каком-либо процессоре начинают расходиться, пострадает только этот процессор.

Итерации в нем уже не сойдутся с заданной точностью. Система же останется работоспособной и перейдет к следующему шагу по времени. Можно предположить, что ошибка при определении решения в небольшой области незначительно ухудшит сходимость метода в целом, поскольку на следующем шаге по времени локальная ошибка будет быстро сглажена. Такое предположение тем более оправдано, что скорость сходимости итераций значительно возрастает при проведении итераций в пределах небольшой области. Предложенный компромисс вполне разумен еще и потому, что на практике такая ситуация при использовании устойчивых схем почти не возникает.

Подход имеет еще и то преимущество, что при сборе информации глобальные данные попадают на процессор, связанный с корневым, по нескольким каналам, обеспечивая их равномерную загрузку.

К недостаткам алгоритма можно отнести невозможность в его рамках произвольно часто собирать результаты счета. Поясним этот факт подробнее, поскольку его игнорирование приводит к трудно идентифицируемым логическим ошибкам.

После запуска программы параллельно будут протекать два процесса:

- синхронный - вычисления на итерациях протекают одновременно на всех процессорах, поскольку перед каждой итерацией процессоры связываются друг с другом, что и позволяет говорить об общем “времени” системы, измеряемом тактами;
- асинхронный - рассылка исходных данных, сбор глобальной информации. После того как транспьютер решетки передал очередную порцию информации на корневой процессор, он продолжает вычисления, независимо от того, получены данные адресатом или нет. В общем случае неизвестно, через сколько тактов корневой транспьютер получит все посланные ему результаты одного временного слоя.

Возможна ситуация, когда процессор, расположенный около корневого, передаст ему результаты (j+1) временного слоя раньше, чем тот получит результаты j слоя от удаленных процессоров. Попытка организовать хранение всех таких преждевременно полученных данных скорее всего не будет иметь успеха. В отсутствие сдерживающего механизма такая ситуация, единожды возникнув, будет повторяться, пока не переполнится вся доступная управляющей программе память. Чтобы избежать подобных неприятностей, можно ***собирать глобальную информацию достаточно “редко”*** (что неконструктивно - понятие “редко” очень сильно зависит от задачи и количества точек, обрабатываемых каждым транспьютером). Можно разрешать ***продолжение расчета*** очередного временного слоя только после того, как корневым процессором полностью получены все данные, относящиеся к предыдущему слою, но это приведет к неоправданному про-

стою транспьютеров решетки во время сбора результатов. Разумнее всего разрешать ***сбор информации*** очередного временного слоя только после того, как полностью получены все данные, относящиеся к предыдущему слою. Рассылать такое разрешение следует в синхронном режиме аналогично передаче признака конца итераций, чтобы обеспечить одновременность его получения всеми процессорами сети. При этом могут возникнуть сложности сбора результатов в наперед заданные моменты времени, однако они легко преодолимы.

Дадим приближенную оценку относительной эффективности предлагаемого алгоритма, по сравнению с традиционным:

$$T_1 = K\,(2R\,T_{link}+T_{calc}),$$

$$T_2 = (K + D)\,T_{calc},$$

где T_{link} - время, необходимое для передачи признака окончания итераций между соседними процессорами; T_{calc} - время, необходимое для расчета одной итерации в пределах одного транспьютера; K - число итераций, необходимых для получения решения с заданной точностью; R - радиус графа, объединяющего транспьютеры; D - диаметр этого графа; T_1, T_2- время, необходимое для получения решения на очередном временном слое по первому способу и при использовании децентрализованного алгоритма, соответственно. Предлагаемый подход эффективен при $T_2 < T_1$, что справедливо при

$$K > \frac{D}{2R}\frac{T_{calc}}{T_{link}}.$$

При $T_{link} \sim T_{calc}$, можно утверждать, что при $K > D/2RL$, где L - число точек, приходящихся на один процессор, использование децентрализованного алгоритма предпочтительнее. В общем случае справедливо соотношение $D/2R<1$, поэтому можно считать, что предлагаемый алгоритм эффективен при $K>L$.

Для 32 транспьютеров, объединенных в решетку 4x8 (рис. 28, только связи, показанные сплошными линиями), $R=6$, $D=10$. При замыкании решетки, например в тор, эти числа можно уменьшить. На рис. 28 (все связи) показан пример топологии, для которой $D=R=4$. Аналогичные графы можно построить для решеток 4x4 ($D=R=3$), 8x8 ($D=R=6$, рис. 29) и т.д.

Для дальнейшего анализа предложенного алгоритма распределенного управления остановимся на вопросах определения ускорения при решении уравнения Лапласа с помощью итерационного алгоритма на параллельной вычислительной системе с распределенной памятью. Будем говорить, что задача имеет размер L, если L - число расчетных точек, покрывающих область, в которой ищется решение.

С учетом изложенного можно приближенно оценить время решения задачи при использовании различных алгоритмов управления:

$$T_0 = KL\tau_c;$$

$$T_c = K\left(\frac{L}{N}\tau_c + \tau_0 + \tau_s\sqrt{\frac{L}{N}} + \tau_0 N\right);$$

$$T_d = \left(K + \sqrt{N}\right)\left(\frac{L}{N}\tau_c + \tau_0 + \tau_s\sqrt{\frac{L}{N}} + \tau_s N\right).$$

Здесь T_0 - время решения задачи на одном процессоре, K - общее число итераций, T_c, T_d - время решения задачи на N процессорах с помощью централизованного и децентрализованного алгоритмов, τ_c - время расчета одной точки на одной итерации, τ_0 - время подготовки данных к передаче, τ_s - время передачи данных, соответствующих одной точке, на соседний процессор.

Время T_c складывается из времени расчета точек (считается, что точки распределены равномерно и на каждый процессор приходится по $\frac{L}{N}$ точек), времени подготовки к обмену данными, времени передачи информации о граничных точках (их число приближенно определяется как корень из числа точек, приходящихся на каждый процессор) и времени сбора глобальной информации. Соотношение приведено в предположении, что во время передачи данных счет останавливается и что линки одного процессора не могут работать параллельно. При таких допущениях оказывается, что независимо от топологии сети, время сбора глобальной информации на корневой транспьютер не может быть меньше, чем $\tau_0 N$, так как необходимо принять сообщение от каждого процессора - всего N сообщений.

Время T_d отличается от T_c общим числом итераций (оно возрастает на величину диаметра графа процессоров) и способом обработки глобальных данных (обмен с корневым транспьютером заменяется на обмен с соседним). Диаметр можно оценить как квадратный корень из числа процессоров. При обмене с соседним процессором необходимо передавать информацию о всех процессорах сети, но это можно делать в одном сообщении, соответственно это займет $\tau_s N$ времени, что более чем на порядок меньше $\tau_0 N$. Эта разница и определяет преимущества **d**-алгоритма.

Запишем ускорения **c**- и **d**-алгоритмов (P_c, P_d) и коэффициенты распараллеливания (K_c, K_d):

$$P_c = \frac{T_0}{T_c}K_c = \frac{P_c}{N};$$

$$P_d = \frac{T_0}{T_d}K_d = \frac{P_d}{N}.$$

На рис. 59 приводятся графики, отражающие типичную зависимость ускорения от числа процессоров при L=100x100. Видно, что

обе кривые (ускорения **c**- и **d**-алгоритмов) ведут себя немонотонно и при определенном числе процессоров N_{max} (различном для разных алгоритмов) достигают максимума при решении задачи фиксированного размера. Это означает, что задачу данного размера невозможно решить с помощью обсуждаемых алгоритмов быстрее, нежели за время, которое потратит система, содержащая $N^*_{c,d}$ процессоров, вне зависимости от того, сколькими процессорами мы располагаем. В связи с этим возникает вопрос: как изменяется максимально возможное ускорение с ростом размера задачи - числа расчетных точек?

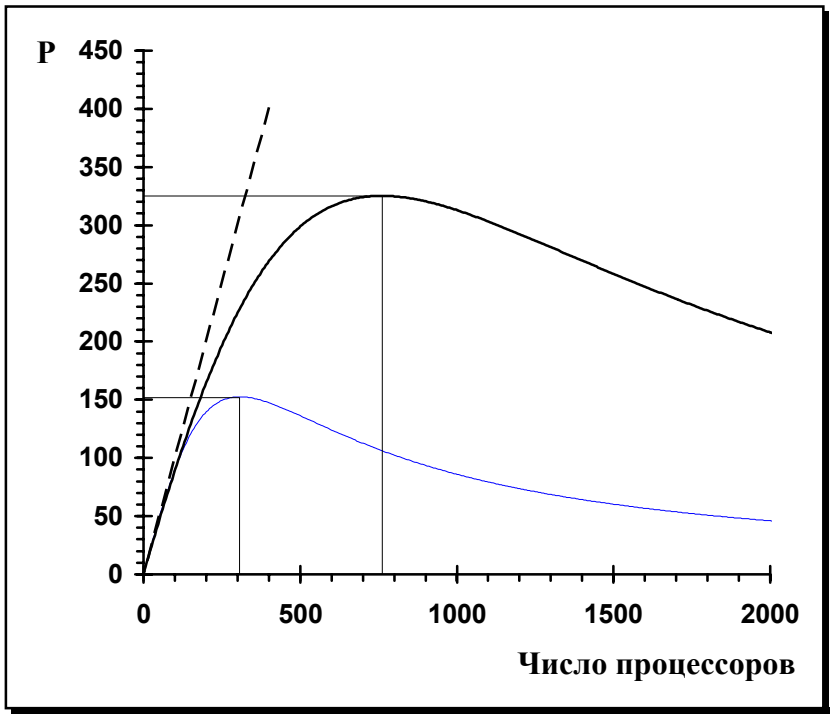

Рис. 59. Зависимость ускорения P_c (——), P_d (——) от числа процессоров при L=10000

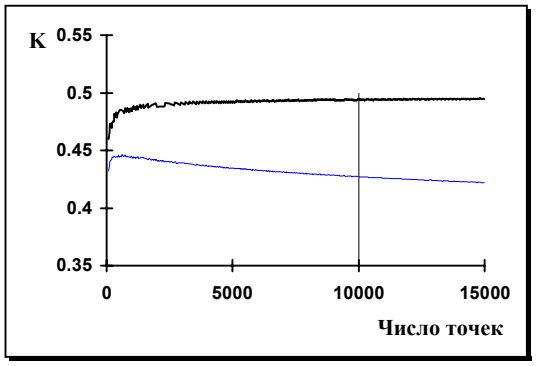

Рис. 60. Максимальный коэффициент распараллеливания *Kc*(——), *Kd*(——)

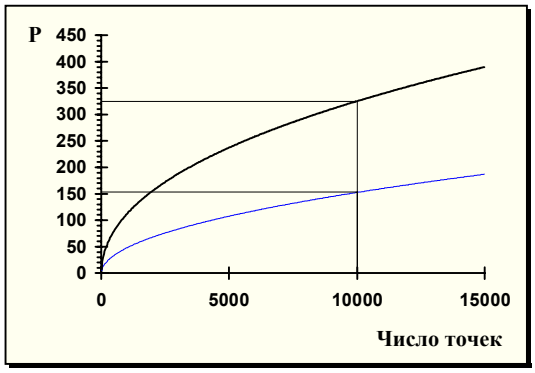

Рис. 61. Максимальное ускорение *Pc* (——), *Pd*(——)

Ускорение растет с ростом размера задачи, причем коэффициент распараллеливания, соответствующий максимальному ускорению, за-

висит от числа расчетных точек очень слабо, что подтверждается рис. 60, 61. Более того, в предположении что $\tau_s < \tau_0$, легко получить следующие оценки для максимального ускорения P_{max} , числа процессоров, при котором оно достигается N_{max} и коэффициента распараллеливания K_{max} , соответствующие **c**-алгоритму:

$$N_{\max} \approx \sqrt{L\frac{\tau_c}{\tau_0}}; \qquad P_{\max} \approx \frac{1}{2}\sqrt{L\frac{\tau_c}{\tau_0}}; \qquad K_{\max} \approx \frac{1}{2}.$$

Можно сделать следующие выводы:

- максимальный коэффициент использования вычислительной мощности K_{max} практически не зависит от рассмотренных параметров параллельной системы и от размера задачи;
- максимальное ускорение растет пропорционально корню квадратному из числа расчетных точек;
- максимальное ускорение растет пропорционально числу процессоров в системе;
- максимальное ускорение **d**-алгоритма превышает (более чем вдвое при L=10000) максимальное ускорение **c**-алгоритма (при подготовке графиков рис. 59-61 использованы значения параметров, полученные при решении двумерного уравнения Лапласа с помощью α–β алгоритма на вычислительной системе, составленной из транспьютеров T800).

Анализ алгоритмов проводился в предположении, что перекрытие во времени вычислительных процессов и процессов передачи данных отсутствует. Это предположение несколько занижает оценки ожидаемой производительности, особенно по отношению к неоднородным системам типа PowerXplorer, в которых появляется возможность разделить вычислительные и передающие процессы не только конкурентно (на транспьютере они конкурируют за ресурс арифметического устройства), но и физически - по разным процессорам, что должно положительно сказаться на производительности системы в целом. Тестовые расчеты подтверждают преимущества **d**-алгоритма перед **c**-алгоритмом даже на небольшом числе процессоров.

Библиотека системы программирования PARIX™

Группы функций

GET_ROOT – получение информации о процессоре

MakeClique, FreeClique, GetClique_Data - виртуальная топология Клика

GetLinkCB – получение управляющего блока виртуального линка

Send, Recv – функции синхронного обмена данными через каналы виртуальных топологий

AInit, ASend, ARecv, ASync, AInfo, AExit – функции асинхронного обмена данными через каналы виртуальных топологий

Select, CondSelect, SelectList, CondSelectList, ReceiveOption, ReceiveOption_B, TimeAfterOption, TimeAfterOption - Селективное ожидание ввода данных или таймаута

TimeNow, TimeWait, TimeAfter – доступ к процессорному таймеру

CreateSem, InitSem, DestroySem, Wait, TestWait, Signal - управление семафорами

Описание функций

При запуске программы указывается количество физических процессоров, выделяемых задаче, и количество запускаемых на них виртуальных процессоров. Поскольку в рамках системы PARIX для программиста отсутствует разница между реальным и виртуальным процессором, под термином процессор имеется в виду виртуальный процессор. Все физические процессоры системы пронумерованы, однако в момент запуска задачи выделенные ей виртуальные процессоры получают свою, независимую от исходной, нумерацию, которой и пользуется программист. Более того, внутри программы нельзя легально определить, сколько процессоров содержит вся система в целом, и какие именно процессоры выделены задаче. В связи с этим, под числом процессоров в дальнейшем понимают число выделенных задаче виртуальных процессоров. Поскольку при запуске задачи под управлением систем PARIX™ запускается столько копий программы, сколько выделено виртуальных процессоров, под номером процессора понимают номер виртуального процессора, на котором запущена соответствующая копия программы.

GET_ROOT

GET_ROOT - макроопределение, предоставляющее доступ к структуре данных, описывающих процессор.

```
#include <epx/root.h>
typedef struct {
    int   MyProcID;  /* номер процессора, начиная с 0 */
    int   MyX;       /* положение процессора на оси x, начиная с 0 */
    int   MyY;       /* положение процессора на оси y, начиная с 0 */
    int   MyZ;       /* положение процессора на оси z, начиная с 0 */
    int   nProcs;  /* общее число процессоров, равное DimX * DimY * DimZ */
    int   DimX;      /* число процессоров по оси x */
    int   DimY;      /* число процессоров по оси y */
    int   DimZ;      /* число процессоров по оси z */
  } RootProc_t;
```

Используя макроопределение ***GET_ROOT()***, можно получить доступ к данным, описывающим общие размеры предоставленной задаче решетки процессоров, положение процессора внутри этой решетки, число предоставленных процессоров и номер используемого процессора.

Пример использования:

```
GET_ROOT ()->ProcRoot->MyProcID  // номер процессора
GET_ROOT ()->ProcRoot->MyX // позиция процессора по оси X
```

MakeClique

```
#include <virt_top.h>       // библиотека: libVT.a
int MakeClique (int reqId,
        int size,
        int xmin, int xmax,
        int ymin, int ymax,
        int zmin, int zmax);
```

MakeClique() объединяет ***size*** процессоров виртуальной топологией *клика*. Каждая пара процессоров *клики* соединена между собой виртуальным каналом. Группа процессоров, образующих *клику*, задается параметрами ***xmin, xmax, ymin, ymax, zmin, zmax***. Вместо ***xmin, ymin, zmin*** может быть указана константа ***MINSLICE,*** соответствующая 0. Вместо ***xmax, ymax, zmax*** - константа ***MAXSLICE***, соответствующая числу процессоров по определяемому направлению. Только процессоры расположенные так, что ***xmin <= x <= xmax, ymin <= y <= ymax и zmin <= z <= zmax*** могут попасть в формируемую *клику*.

Если вместо ***size*** указано ***MAXCLIQUE***, формируется *клика* максимально возможного размера. Целое число ***reqId*** должно быть одинаковым для всех образующих *клику* процессоров.

MakeClique() возвращает идентификатор построенной топологии. Этот идентификатор и номер линка определяют конкретный логический линк, связывающий данный процессор с некоторым другим. Линки пронумерованы от 0 до ***size***-1. Логический линк ***i*** соответствует узлу с номером ***i*** внутри *клики*.

В случае ошибки ***MakeClique***() возвращает код < 0.

При вызове подпрограммы ***MakeClique*** могут возникнуть следующие ошибки:

EINPART - неправильно указаны параметры, определяющие используемый для создания топологии раздел.

ENOTEPROCS -заданное число процессоров не может быть выделено. Используемый раздел не содержит достаточного количества процессоров.

Функции создания других топологий:

MakePipe, MakeRing, MakeStar, Make2DGrid, Make3DGrid, Make2DTorus, Make3DTorus, MakeHCube, MakeTree and MakeDeb

FreeClique

```
#include <virt_top.h>      // библиотека: libVT.a
int FreeClique (int topId);
```

FreeClique() освобождает данные, соответствующие образованной ранее топологии типа *клика*. Аргумент ***topId*** – идентификатор топологии, возвращенный функцией ***MakeClique***().

В случае ошибки подпрограмма ***FreeClique***() возвращает код < 0, иначе 0.

В случае ошибки переменная ***errno*** может содержать значение

EINVAL - указан неверный аргумент, не являющийся дескриптором созданной ранее виртуальной топологии типа *клика*.

GetClique_Data

```
#include <virt_top.h>  // библиотека: libVT.a
CliqueData_t *GetClique_Data (int topId)
```

GetClique_Data() - возвращает указатель на структуру описания топологии типа *клика*. Аргумент ***topId*** – идентификатор топологии, возвращенный функцией ***MakeClique***(). В случае ошибки возвращается значение ***NULL***. Вид возвращаемой структуры определен в файле ***<virt_top.h>***.

```
struct CliqueData_t {
    char type; /* тип топологии */
    int  status;/* статус процессора */
    int  id;            /* идентификатор процессора */
    int  size; /* число процессоров, объединенных топологией */
};
```

Топологии *клика* соответствует константа ***type = CLIQUE_TYPE***.

Если процессор не является частью указанной *клики*, поле ***status*** будет содержать значение ***CLIQUE_NONE***. В противном случае, поле ***status*** будет содержать значение ***CLIQUE_IN***.

Каждому процессору, входящему в *клику*, соответствует номер ***id***, который может иметь значение из диапазона: ***0 <= id < size***. Каждый из процессоров *клики* связан с остальными посредством ***size-1*** линков. Линк с номером ***i != id*** связывает процессор с номером ***id*** с процессором, имеющим номер ***i*** внутри данной клики.

В случае ошибки подпрограмма ***GetClique_Data***() возвращает ***NULL***, иначе адрес структуры данных, описывающей топологию.

В случае ошибки переменная ***errno*** может содержать значение ***EINVAL,*** что соответствует неверному заданию аргумента ***topId***.

Send

```
#include <epx/comm.h>
   int Send (int TopId, int LogLinkId, void *Data, int Size)
```

Send() синхронно передает ***Size*** байт данных, адрес которых задан указателем ***Data*** через линк ***LogLinkId*** виртуальной топологии ***TopId.***

Если принимающий процессор ожидает меньше данных, чем посылает передающий процессор, функция ***Send*** возвращает число ожидаемых на приемном конце байт.

Send() возвращает следующие значения:

>= 0 число переданных байт

< 0 код ошибки установлен в переменной ***errno*:**

EINVAL в случае, если параметр ***LogLinkId*** вне допустимого диапазона, либо контрольный блок соответствующего логического линка содержит пустое значение.

Другие возможные значения соответствуют кодам возврата функции ***, SendLink().***

Recv

#include <epx/comm.h>
int Recv (int TopId, int LogLinkId, void *Data, int Size);

Recv() принимает через линк ***LogLinkId*** виртуальной топологии ***TopId*** данные, объемом ***Size*** байт, и записывает их в область памяти, адрес которой определяется указателем ***Data.***

Если принимается меньше данных, чем передается, то функция ***Recv()*** возвращает код ошибки.

Recv() возвращает следующие значения:
>= 0 число принятых байт,
< 0 код ошибки установлен в переменной ***errno*****:**
EINVAL в случае, если параметр ***LogLinkId*** вне допустимого диапазона, либо контрольный блок соответствующего логического линка содержит пустое значение.
Другие возможные значения соответствуют кодам возврата функции ***RecvLink().***

AInit

#include <epx/comm.h>
int AInit (int TopId, int Threads, int Size);

Подпрограмма ***AInit()*** инициализирует параметры, используемые подпрограммами ***ASend()*** и ***ARecv()*** или, при повторном вызове, устанавливает параметры в новые значения***.*** ***TopId*** указывает ранее созданную виртуальную топологию, ***Threads*** указывает максимальное число тредов, которые могут быть использованы для асинхронной передачи данных, ***Size*** определяет максимальный объем памяти, используемой при обменах. Подпрограмма ***Aexit()*** или ***Freetop()*** должны быть вызваны для остановки тредов, запущенных для выполнения асинхронных передач данных.

Size = -1 - нет ограничения на используемый объем памяти***.***
Size = 0 передаваемые данные не копируются во временный буфер***.***
Threads = -1 - нет ограничения на число используемых для обмена тредов***.***

AInit() возвращает следующие значения:
0 инициализация успешно выполнена
< 0 код ошибки установлен в переменной ***errno*****:**

EINVAL неправильное значение ***TopId,*** или ***Threads*** либо ***Size*** вне допустимого диапазона
ENOMEM недостаточно оперативной памяти

ASend

#include <epx/comm.h>
int ASend (int TopId, int LogLinkId,
byte *Data, int Size, int *Result)

Подпрограмма ***ASend()*** применяется для асинхронной передачи ***Size*** байт данных, адрес которых определяется указателем ***Data*** через линк ***LogLinkId*** виртуальной топологии ***TopId***. Число переданных байт возвращается через переменную ***Result.***

ASend() возвращает следующие значения:
0 передающий тред успешно запущен.
-2 все треды уже используются, следует вызвать ***AInit*** снова с параметром ***Threads = Threads + 1*** или ***Threads = -1.***
-1 код ошибки установлен в переменной ***errno*****:**
EINVAL в случае, если параметр ***LogLinkId*** вне допустимого диапазона, либо указатель на контрольный блок соответствующего логического линка содержит пустое значение.
EAGAIN нет готовых к работе тредов
ENOMEM недостаточно оперативной памяти
Другие значения соответствует кодам возврата функций ***StartThread*** или ***SendLink.***

ARecv

#include <epx/comm.h>
int ARecv (int TopId, int LogLinkId,
byte *Data, int Size, int *Result);

Подпрограмма ***ARecv()*** применяется для асинхронного приема ***Size*** байт данных, через линк ***LogLinkId*** виртуальной топологии ***TopId*** и записи их в область памяти определяемым указателем ***Data***. Число принятых байт возвращается через переменную ***Result.***

ARecv() возвращает следующие значения:
0 принимающий тред успешно запущен.
-2 все треды уже используются, следует вызвать ***AInit*** снова с параметром ***Threads = Threads + 1*** или ***Threads = -1.***

-1 код ошибки установлен в переменной ***errno.***

EINVAL в случае, если параметр ***LogLinkId*** вне допустимого диапазона, либо указатель на контрольный блок соответствующего логического линка содержит пустое значение.

EAGAIN нет готовых к работе тредов

ENOMEM недостаточно оперативной памяти

Другие значения соответствует кодам возврата функций ***StartThread*** или ***RecvLink.***

ASync

#include <epx/comm.h>
int ASync (int TopId, int LogLinkId);

ASync() ожидает завершения обмена данными через линк ***LogLinkId*** или через все линки топологии ***TopId*** (при ***LogLinkId = -1***)**.** На время ожидания процесс, вызвавший функцию ***ASync()***, блокируется.

ASync() возвращает следующие значения:

0 принимающий тред успешно запущен.

-1 код ошибки установлен в переменной ***errno.***

EINVAL неправильное значение ***TopId,*** или ***Threads*** вне допустимого диапазона

ESRCH ошибка выполнения функции ***WaitThread().***

AInfo

#include <epx/comm.h>
int AInfo (int TopId, int LogLinkId);

С помощью функции ***AInfo()*** можно определить число незавершенных обменов через линк ***LogLinkId*** или через все линки топологии ***TopId*** (при ***LogLinkId = -1***)**.**

AInfo() возвращает следующие значения:

>= 0 число незавершенных обменов

<0 код ошибки установлен в переменной ***errno***:

EINVAL неправильное значение ***TopId,*** или ***Threads*** вне допустимого диапазона.

AExit

#include <epx/comm.h>
int AExit (int TopId);

AExit() ожидает завершения всех обменов, останавливает треды и освобождает структуры данных, используемые топологией ***TopId.*** Для завершения тредов необходимо вызвать либо функцию ***AExit(),*** либо функцию ***Freetop().***

AExit() возвращает следующие значения:
0 в случае успешного завершения
<0 ошибка.

Select

#include <epx/select.h>
int Select (int nOpts, Option_t Opt1, Option_t Opt2, ...);

Select() блокирует выполнение программы до тех пор, пока на одном из указанных ***nOpts*** расширенных каналов ***Opt1, Opt2*** не появится сообщение. Расширенным каналом может служить любой виртуальный канал, с которого ожидается сообщение. Кроме того, расширенным каналом может служить таймер. В этом случае под сообщением понимают наступление заданного при инициализации данного расширенного канала момента времени.

Функция ***Select()*** возвращает номер одного из перечисленных в ее параметрах расширенных каналов, на который уже пришло сообщение. Если к моменту вызова ***Select()*** сообщения пришли на несколько расширенных каналов, возвращается номер первого по списку канала вне зависимости от того, какое из сообщений пришло раньше других.

Для инициализации расширенного канала используются функции ***ReceiveOption(), TimeAfterOption(), ReceiveOption_B()*** и ***TimeAfterOption_B().***

Если расширенный канал соответствует виртуальному линку, его готовность означает, что ввод данных с помощью функции типа ***RecvLink()*** с соответствующего линка может быть осуществлен без дополнительного ожидания.

Если расширенный канал соответствует таймеру (инициализирован с помощью одной из функций ***TimeAfterOption()*** или

TimeAfterOption_B()), то его готовность означает, что в заданный промежуток времени сообщений с других каналов не поступило.

CondSelect

#include <epx/select.h>
int CondSelect (int nOpts, Option_t Opt1, Option_t Opt2, ...);

Функция ***CondSelect()*** полностью аналогична функции ***Select()*** с той разницей, что она не блокирует выполнение вызвавшего ее процесса, а немедленно возвращает значение -1, если ни на один из расширенных каналов сообщение еще не пришло.

SelectList

#include <epx/select.h>
int SelectList (int nOpts, Option_t Options[]);

Функция ***SelectList()*** полностью аналогична функции ***Select()*** с той разницей, что список расширенных каналов задается с помощью массива, в котором перечисляются ***nOpts*** расширенных каналов.

CondSelectList

#include <epx/select.h>
int CondSelectList(int nOpts, Option_t Options[]);

Функция ***CondSelectList()*** полностью аналогична функции ***CondSelect()*** с той разницей, что она не блокирует выполнение вызвавшего ее процесса, а немедленно возвращает значение -1, если ни на один из расширенных каналов сообщение еще не пришло.

ReceiveOption

#include <epx/select.h>
Option_t ReceiveOption (LinkCB_t *Link);
Функция ***ReceiveOption()*** возвращает инициализированную в соответствии с указателем ***Link*** на контрольный блок виртуального канала структуру ***Option_t***. Получить значение указателя на контрольный блок виртуального канала какой-либо виртуальной топологии можно с помощью функции ***GetLinkCB.***

ReceiveOption_B

#include <epx/select.h>

Option_t ReceiveOption_B (LinkCB_t *Link, int guard);

Функция ***ReceiveOption_B()*** аналогична функции ***ReceiveOption()*** с той разницей, что позволяет указать при инициализации необходимость ожидания ввода с указанного расширенного канала. При ***guard=0*** соответствующий расширенный канал игнорируется в списке расширенных каналов, при ***guard!=0*** расширенный канал обрабатывается обычным образом.

TimeAfterOption

#include <epx/select.h>
Option_t TimeAfterOption (time_t EndTime);

Функция ***TimeAfterOption()*** возвращает инициализированную в соответствии с временем ***EndTime*** структуру ***Option_t.*** В качестве параметра ***EndTime*** следует указывать некоторый момент времени. Например, если требуется задать интервал времени в 1 с, то следует использовать ***EndTime=TimeNow()+CLOCK__TICK.***

TimeAfterOption_B

#include <epx/select.h>
Option_t TimeAfterOption_B (time_t EndTime, int guard)

Функция ***TimeAfterOption_B()*** аналогична функции ***TimeAfterOption()*** с той разницей, что позволяет указать при инициализации необходимость ожидания ввода с указанного расширенного канала. При ***guard=0*** соответствующий расширенный канал игнорируется в списке расширенных каналов, при ***guard!=0*** расширенный канал обрабатывается обычным образом.

GetLinkCB

#include <epx/topology.h>
LinkCB_t *GetLinkCB (int TopId, int LogLinkId, int *Error);

Функция ***GetLinkCB*** возвращает указатель на контрольный блок линка. В случае ошибки возвращается значение ***NULL,*** код ошибки передается в этом случае через переменную ***Error. TopId*** - задает созданную ранее виртуальную топологию, ***LogLinkId*** - задает номер виртуального линка в пределах этой топологии.

Возможные значения ***Error:***

EINVAL неправильное значение ***TopId,*** или ***LogLinkId*** вне допустимого диапазона.

TimeNow

#include <epx/time.h>
unsigned int TimeNow (void);

Функция возвращает текущее значение процессорного таймера, выраженное в условных единицах - «тиках». Число «тиков» в секунде определяется константой ***CLOCK__TICK***. Примерно каждую 71 минуту значение таймера становится равным 0.

TimeWait

#include <epx/time.h>
void TimeWait (unsigned int Time);

Если ***Time-TimeNow()<=INT_MAX***, то процесс, вызвавший подпрограмму ***TimeWait***, задерживается до тех пор, пока не наступит время, указанное параметром ***Time***. В противном случае выполнение процесса продолжается без задержки. Нельзя задать интервал ожидания, превышающий 36 минут.

TimeAfter

#include <epx/time.h>
int TimeAfter (unsigned int Time1, unsigned int Time2);

Функция возвращает положительное значение, если время ***Time1*** больше, чем ***Time2*** (***Timet1-Time2<=INT_MAX***). В противном случае возвращаемое значение равно ***0***.

CreateSem

SYNOPSIS

#include <epx/sem.h>
Semaphore_t CreateSem (int Count);

Создание семафора и его инициализация значением ***Count***.

При успешном создании семафора возвращается его значение. В случае ошибки создания семафора возвращается значение ***NULL*** и устанавливается переменная ***errno***.

DestroySem

int DestroySem (Semaphore_t *Sem);

Уничтожение семафора, созданного с помощью функции ***CreateSem***. Эту подпрограмму необходимо вызвать для корректного

освобождения ресурсов операционной системы, используемых семафором. При успешном освобождении ресурсов возвращается значение 0, иначе возвращается код ошибки и устанавливается переменная ***errno***.

Wait

void Wait (Semaphore_t *Sem);

Блокирующая проверка семафора. Если значение семафора равно 0, то выполнение процесса задерживается до тех пор, пока семафор не будет освобожден. Если значение семафора положительно, оно уменьшается на 1 и выполнение процесса продолжается.

TestWait

int TestWait (Semaphore_t *Sem);

Не блокирующая проверка семафора. Действие этой функции отличается от действия функции ***TestWait()*** тем, что в том случае, когда значение семафора равно 0, сразу возвращается значение ***FALSE***. Если значение семафора положительно, оно уменьшается на 1 и возвращается значение ***TRUE***. В любом случае выполнение процесса не задерживается.

Signal

void Signal (Semaphore_t *Sem);

Освобождение семафора. При наличии процессов, ожидающих освобождения семафора, один из них начинает выполняться, иначе значение семафора увеличивает на 1.

Перечень функций системы PARIX™

Предметный указатель