Параллельные высокопроизводительные системы

Сферы, требующие высокой производительности:

- Атомная энергетика
- Авиастроение
- ракетно-космические технологии
- автомобилестроение
- нефте- и газодобыча
- фармакология
- прогноз погоды и моделирование изменения климата
- сейсморазведка
- проектирование электронных устройств
- синтез новых материалов

Классы компьютеров

Персональные компьютеры.

Как правило, в этом случае подразумеваются однопроцессорные системы на платформе Intel или AMD, работающие под управлением однопользовательских операционных систем (Microsoft Windows и др.). Используются в основном как персональные рабочие места.

Рабочие станции.

Это чаще всего компьютеры с RISC процессорами с многопользовательскими операционными системами, относящимися к семейству ОС UNIX. Содержат от одного до четырех процессоров. Поддерживают удаленный доступ. Могут обслуживать вычислительные потребности небольшой группы пользователей.

Суперкомпьютеры.

Отличительной особенностью суперкомпьютеров является то, что это, как правило, большие и, соответственно, чрезвычайно дорогие многопроцессорные системы. В большинствеслучаев в суперкомпьютерах используются те же серийно выпускаемые процессоры, что и в рабочих станциях. Поэтому зачастую различие между ними не столько качественное, сколько количественное. Например, можно говорить о 4-х процессорной рабочей станции фирмы SUN и о 64-х процессорном суперкомпьютере фирмы SUN. Скорее всего, в том и другом случае будут использоваться одни и те же микропроцессоры.

Кластерные системы.

В последние годы широко используются во
всем мире как дешевая альтернатива суперкомпьютерам. Система требуемой производительности собирается из готовых серийно выпускаемых компьютеров, объединенных опять же с помощью некоторого серийно выпускаемого коммуникационного оборудования.

Единицы измерения производительности вычислительных систем

1 Mflops (мегафлопс) = 1 миллион оп/сек
1 Gflops (гигафлопс) = 1 миллиард оп/сек
1 Tflops (терафлопс) = 1 триллион оп/сек

История

Сегодня параллелизмом в архитектуре компьютеров уже мало кого удивишь. Все современные микропроцессоры, будь то Pentium III или PA-8700, MIPS R14000, E2K или Power3 используют тот или иной вид параллельной обработки. В ядре Pentium 4 на разных стадиях выполнения может одновременно находиться до 126 микроопераций. На презентациях новых чипов и в пресс-релизах корпораций это преподносится как последнее слово техники и передовой край науки, и это действительно так, если рассматривать реализацию этих принципов в миниатюрных рамках одного кристалла.

Вместе с тем, сами эти идеи появились очень давно. Изначально они внедрялись в самых передовых, а потому единичных, компьютерах своего времени. Затем после должной отработки технологии и удешевления производства они спускались в компьютеры среднего класса, и наконец сегодня все это в полном объеме воплощается в рабочих станциях и персональных компьютерах.

Для того чтобы убедиться, что все основные нововведения в архитектуре современных процессоров на самом деле используются еще со времен, когда ни микропроцессоров, ни понятия суперкомпьютеров еще не было, совершим маленький экскурс в историю, начав практически с момента рождения первых ЭВМ.

IBM 701 (1953), IBM 704 (1955): разрядно-параллельная память, разрядно-параллельная арифметика.
Все самые первые компьютеры (EDSAC, EDVAC, UNIVAC) имели разрядно-последовательную память, из которой слова считывались последовательно бит за битом. Первым коммерчески доступным компьютером, использующим разрядно-параллельную память (на CRT) и разрядно-параллельную арифметику, стал IBM 701, а наибольшую популярность получила модель IBM 704 (продано 150 экз.), в которой,

помимо сказанного, была впервые применена память на ферритовых сердечниках и аппаратное АУ с плавающей точкой.

IBM 709 (1958): независимые процессоры ввода/вывода.
Процессоры первых компьютеров сами управляли вводом/выводом. Однако скорость работы самого быстрого внешнего устройства, а по тем временам это магнитная лента, была в 1000 раз меньше скорости процессора, поэтому во время операций ввода/вывода процессор фактически простаивал. В 1958г. к компьютеру IBM 704 присоединили 6 независимых процессоров ввода/вывода, которые после получения команд могли работать параллельно с основным процессором, а сам компьютер переименовали в IBM 709. Данная модель получилась удивительно удачной, так как вместе с модификациями было продано около 400 экземпляров, причем последний был выключен в 1975 году - 20 лет существования!

IBM STRETCH (1961): опережающий просмотр вперед, расслоение памяти.
В 1956 году IBM подписывает контракт с Лос-Аламосской научной лабораторией на разработку компьютера STRETCH, имеющего две принципиально важные особенности: опережающий просмотр вперед для выборки команд и расслоение памяти на два банка для согласования низкой скорости выборки из памяти и скорости выполнения операций.

ATLAS (1963): конвейер команд.
Впервые конвейерный принцип выполнения команд был использован в машине ATLAS, разработанной в Манчестерском университете. Выполнение команд разбито на 4 стадии: выборка команды, вычисление адреса операнда, выборка операнда и выполнение операции. Конвейеризация позволила уменьшить время выполнения команд с 6 мкс до 1,6 мкс. Данный компьютер оказал огромное влияние, как на архитектуру ЭВМ, так и на программное обеспечение: в нем впервые использована мультипрограммная ОС, основанная на использовании виртуальной памяти и системы прерываний.

CDC 6600 (1964): независимые функциональные устройства.
Фирма Control Data Corporation (CDC) при непосредственном участии одного из ее основателей, Сеймура Р.Крэя (Seymour R.Cray) выпускает компьютер CDC-6600 - первый компьютер, в котором использовалось несколько независимых функциональных устройств. Машина имела громадный успех на научном рынке, активно вытесняя машины фирмы IBM. 2-3 Mflops

CDC 7600 (1969): конвейерные независимые функциональные устройства. CDC выпускает компьютер CDC-7600 с восемью независимыми конвейерными функциональными устройствами - сочетание параллельной и конвейерной обработки. 10-15 Mflops

ILLIAC IV (1974): матричные процессоры.
Проект: 256 процессорных элементов (ПЭ) = 4 квадранта по 64ПЭ, возможность реконфигурации: 2 квадранта по 128ПЭ или 1 квадрант из 256ПЭ, такт 40нс, производительность 1Гфлоп;
работы начаты в 1967 году, к концу 1971 изготовлена система из 1 квадранта, в 1974г. она введена в эксплуатацию, доводка велась до 1975 года;
центральная часть: устройство управления (УУ) + матрица из 64 ПЭ;
УУ это простая ЭВМ с небольшой производительностью, управляющая матрицей ПЭ; все ПЭ матрицы работали в синхронном режиме, выполняя в каждый момент времени одну и ту же команду, поступившую от УУ, но над своими данными;

ПЭ имел собственное АЛУ с полным набором команд, ОП - 2Кслова по 64 разряда, цикл памяти 350нс, каждый ПЭ имел непосредственный доступ только к своей ОП;
сеть пересылки данных: двумерный тор со сдвигом на 1 по границе по горизонтали;
Несмотря на результат в сравнении с проектом: стоимость в 4 раза выше, сделан лишь 1 квадрант, такт 80нс, реальная произв-ть до 50Мфлоп - данный проект оказал огромное влияние на архитектуру последующих машин, построенных по схожему принципу, в частности: PEPE, BSP, ICL DAP.

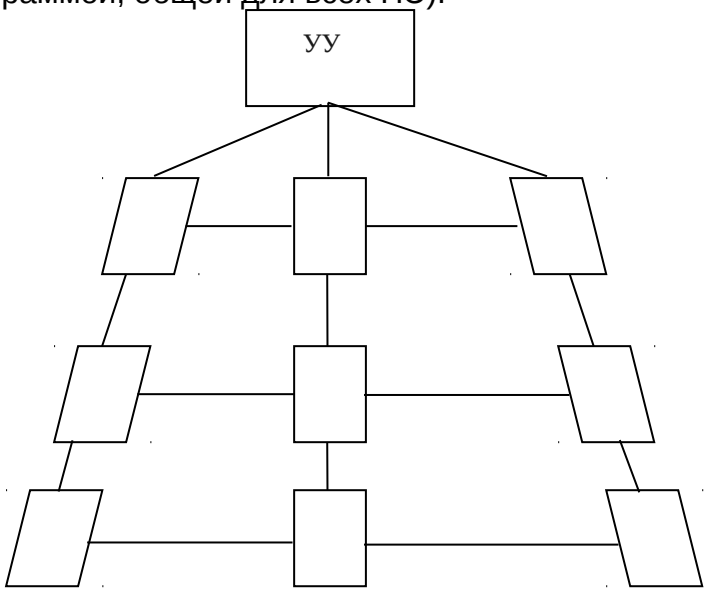
CRAY 1 (1976): векторно-конвейерные процессоры
В 1972 году С.Крэй покидает CDC и основывает свою компанию Cray Research, которая в 1976г. выпускает первый векторно-конвейерный компьютер CRAY-1: время такта 12.5нс, 12 конвейерных функциональных устройств, пиковая производительность 160 миллионов операций в секунду, оперативная память до 1Мслова (слово - 64 разряда), цикл памяти 50нс. Главным новшеством является введение векторных команд, работающих с целыми массивами независимых данных и позволяющих эффективно использовать конвейерные функциональные устройства.

www.top500.org

Первое место на 06.2006 г. Занимает BlueGene/L - eServer Blue Gene Solution IBM. Производительность порядка 300 Tflops. 32768 GB. 131072 процессоров.
В этом же списке суперкомпьютеры компании NEC, Fujitsu, DELL, SGI, Cray, Sun, Hewlett-Packard.

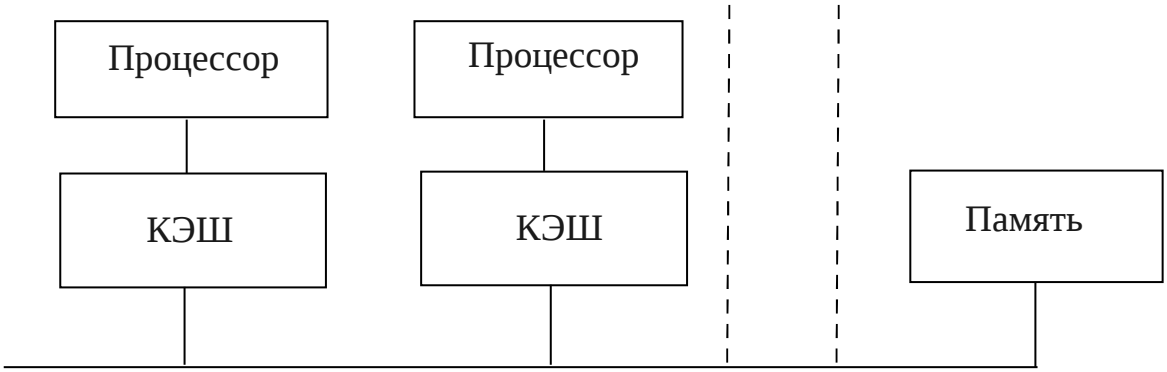
Архитектуры многопроцессорных систем (МВС)

Матричные суперкомпьютеры
Состоят из процессорных элементов (ПЭ): ЦПУ, память, система коммуникации. Обмен данными между процессорами осуществлялся через специальную матрицу коммуникационных каналов.
Первые матричные МВС выпускались буквально поштучно, поэтому их стоимость была фантастически высокой. Серийные же образцы подобных систем, такие как ICL DAP, включавшие до 8192 ПЭ, появились значительно позже, однако не получили широкого распространения ввиду сложности программирования МВС с одним потоком управления (с одной программой, общей для всех ПЭ).



Многопроцессорные системы с массовым параллелизмом (MPP – Massively Parallel Processing)
Характризуются распределенной памятью и произвольной коммуникационной системой между ПЭ. При этом, как правило, каждый из ПЭ МРР системы является универсальным процессором, действующим по своей собственной программе (в отличие от общей программы для всех ПЭ матричной МВС).

Симметричные мультипроцессорные системы (Symmetric Multi-Processing – SMP)
В подобных компьютерах объединялось от 2 до 16 процессоров, которые имели равноправный (симметричный) доступ к общей оперативной памяти.



SMP архитектура обладает весьма ограниченными возможностями по наращиванию числа процессоров в системе из-за резкого увеличения числа конфликтов при обращении к общей шине памяти.

В связи с этим оправданной представлялась идея снабдить каждый процессор собственной оперативной памятью, превращая компьютер в объединение независимых вычислительных узлов.

Такой подход значительно увеличил степень масштабируемости многопроцессорных систем, но в свою очередь потребовал разработки специального способа обмена данными между вычислительными узлами,реализуемого обычно в виде механизма передачи сообщений (Message Passing). Компьютеры с такой архитектурой являются наиболее яркими представителями MPP систем. В настоящее время эти два направления (или какие-то их комбинации) являются доминирующими в развитии суперкомпьютерных технологий.

NUMA-архитектура (Non Uniform Memory Access)
Нечто среднее между SMP и MPP. Память физически разделена, но логически общедоступна. При этом время доступа к различным блокам памяти становится неодинаковым. В одной из первых систем этого типа Cray T3D время доступа к памяти другого процессора было в 6 раз больше, чем к своей собственной.

В настоящее время развитие суперкомпьютерных технологий идет по четырем основным направлениям: векторно-конвейерные суперкомпьютеры, SMP системы, MPP системы и кластерные системы.

Типы процессоров

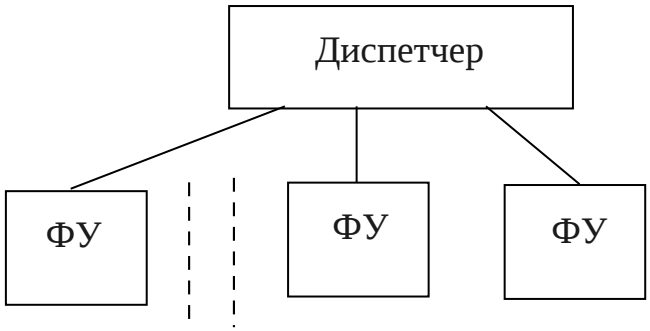
Суперскалярные

Суперскалярная архитектура центрального процессора реализует параллелизм на единственном чипе, таким образом позволяя системе работать намного быстрее, чем это иначе было бы на той же частоте. Суперскалярная архитектура извлекает, выполняет, и возвращает результаты больше чем одной стандартной инструкции в течение одного такта.

Самые простые процессоры - скалярные процессоры. Скалярный процессор обрабатывает один элемент данных одновременно. В векторном процессоре, в отличие от этого, единственная инструкция работает одновременно с несколькими элементами данных. Различие походит на различие между скалярной и векторной арифметикой. Суперскалярный процессор смесь этих двух. Одну инструкцию обрабатывает одино функциональное устройство, но таких устройств несколько, таким образом за один такт может буть обработано несколько инструкций.

Суперскалярный процессор обычно выполненяет более одной инструкции за цикл машины. Правда обработка несколькоих инструкций за раз не делает архитектуру суперскалярной. Простая конвейерная обработка, где центральный процессор может загружать инструкцию, выполнять предыдущую и сохранять препредыдущую (таким образом выполняется 3 инструкций в одно и то же время) не будет суперскалярной обработкой.

В суперскалярном центральном процессоре, есть несколько функциональных устройств одно типа, наряду с дополнительной схемой, чтобы распределять инструкции между ними. Например, большинство суперскалярных процессоров включает более чем одно арифметико-логическое устройство (для целочисленных вычислений). Диспетчер читает инструкции из памяти и решает, которые из них можно выполнять параллельно, посылая их функциональным устройствам.



Работа диспетчера является ключевой в целом для суперскалярной архитектуры. Следует заметить, что решаемые им проблемы не являются тривиальными. Инструкции a = b + c; d = e + f можно выполнять параллельно, потому что ни один из результатов не зависит от других вычислений. Однако, инструкции a = b + c; d = a + f может могут, а может

и не могут выполняться параллельно, в зависимости от порядка, в котором заканчивается их обработка.

В современных разработках стараются увеличить точность работы диспетчера с тем, чтобы загрузить все фунциональные устройства ЦПУ. Это стало особенно важным, с увеличением числа функциональных усройств. В то время как ранние суперскалярные центральные процессоры имели бы два ALU и один FPU, современный проект такой как PowerPC 970 включают два процессора по четыре ALU и два FPU в каждом. Если диспетчер будет неэффективен, то серьезно пострадает производительность системы в целом.

CDC-6600 Сеймура Крея с 1965 часто упоминается как первый суперскалярный проект. Микропроцессоры Intel i960CA (1988) и 29050 (1990) линейки AMD-29000 был первыми коммерческими микропроцессорами суперскалярного типа на одном чипе. RISC-процессоры принесли суперскалярное понятие на микрокомпьютеры, потому что RISC-архитектура приводит к простому ядру, позволяя прямую отправку инструкции и подключение к обработке нескольких функциональных устройств (типа ALU). Это было причиной, что RISC-процессоры были быстрее чем CISC в течение 1980-ых и 1990-ых.

За исключением центральных процессоров, используемых в некоторых переносных устройствах, по существу все центральные процессоры общего назначения, развитые приблизительно с 1998 были суперскалярными, начиная с "P6" (Pentium Pro и Pentium II). В Intel набор команд CISC реализовывался на суперскалярной микроархитектуре RSIC. Сложные инструкции внутри приводились к RISC-подобным, позволяя процессору использовать преимущества, лежащие в основе процессора, оставаясь совместимыми с более ранними процессорами Intel.

Серьезные усовершенствования в качестве блока управления теперь кажутся маловероятными, ограничивая будущие усовершенствования скорости суперскалярной архиекуры. Одно потенциальное решение этой проблемы состоит в том, чтобы переместить логику диспетчера из чипа в компилятор, который может потратить значительно больше времени и усилий на принятии лучших возможных решений. Это - основная предпосылка очень длинного слова инструкции (VLIW) архитектуры центрального процессора, который также известен как статический суперскаляр или планирование при компиляции.

Векторно-конвейерные суперкомпьютеры

Первый векторно-конвейерный компьютер Cray-1 появился в 1976 году. Архитектура его оказалась настолько удачной, что он положил начало целому семейству компьютеров. Название этому семейству компьютеров дали два принципа, заложенные в архитектуре процессоров:

- конвейерная организация обработки потока команд
- введение в систему команд набора векторных операций, которые

позволяют оперировать с целыми массивами данных.

Длина одновременно обрабатываемых векторов в современных векторных компьютерах составляет, как правило, 128 или 256 элементов.

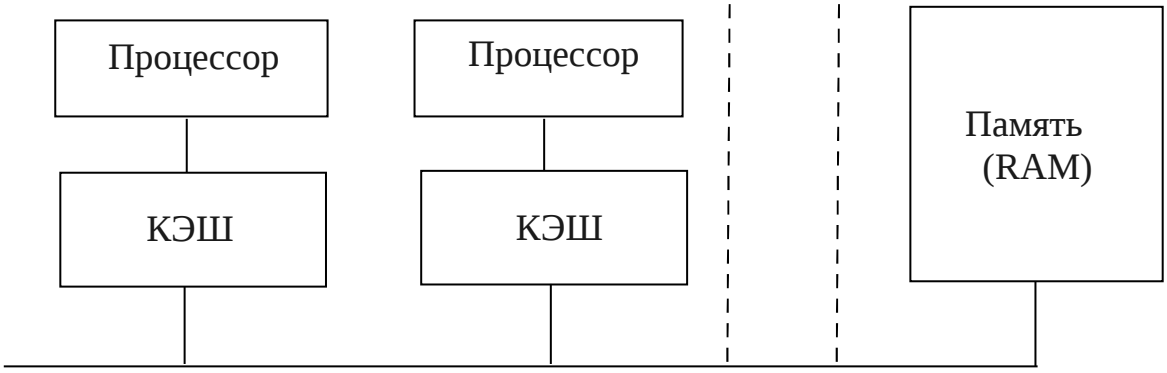
Очевидно, что векторные процессоры должны иметь гораздо более сложную структуру и по сути дела содержать множество арифметических устройств. Основное назначение векторных операций состоит в распараллеливании выполнения операторов цикла, в которых в основном и сосредоточена большая часть вычислительной работы.

Для этого циклы подвергаются процедуре векторизации с тем, чтобы они могли реализовываться с использованием векторных команд. Как правило, это выполняется автоматически компиляторами при изготовлении ими исполнимого кода программы. Поэтому векторно-конвейерные компьютеры не требовали какой-то специальной технологии программирования, что и явилось решающим фактором в их успехе на компьютерном рынке. Тем не менее, требовалось соблюдение некоторых правил при написании циклов с тем, чтобы компилятор мог их эффективно векторизовать.

Исторически это были первые компьютеры, к которым в полной мере было применимо понятие суперкомпьютер. Как правило, несколько векторно-конвейерных процессоров (2-16) работают в режиме с общей памятью (SMP), образуя вычислительный узел, а несколько таких узлов объединяются с помощью коммутаторов, образуя либо NUMA, либо MPP систему. Типичными представителями такой архитектуры являются компьютеры CRAY J90/T90, CRAY SV1, NEC SX-4/SX-5. Уровень развития микроэлектронных технологий не позволяет в настоящее время производить однокристальные векторные процессоры, поэтому эти системы довольно громоздки и чрезвычайно дороги. В связи с этим, начиная с середины 90-х годов, когда появились достаточно мощные суперскалярные микропроцессоры, интерес к этому направлению был в значительной степени ослаблен.

Суперкомпьютеры с векторно-конвейерной архитектурой стали проигрывать системам с массовым параллелизмом. Однако в марте 2002 г. корпорация NEC представила систему Earth Simulator из 5120 векторно-конвейерных процессоров, которая в 5 раз превысила производительность предыдущего обладателя рекорда – MPP системы ASCI White из 8192 суперскалярных микропроцессоров. Это, конечно же, заставило многих по-новому взглянуть на перспективы векторно-конвейерных систем.

Симметричные мультипроцессорные системы (SMP)



Характерной чертой многопроцессорных систем SMP архитектуры является то, что все процессоры имеют прямой и равноправный доступ к любой точке общей памяти. Первые SMP системы состояли из нескольких однородных процессоров и массива общей памяти, к которой процессоры подключались через общую системную шину. Однако очень скоро обнаружилось, что такая архитектура непригодна для создания сколь либо масштабных систем. Первая возникшая проблема – большое число конфликтов при обращении к общей шине. Остроту этой проблемы удалось частично снять разделением памяти на блоки, подключение к которым с помощью коммутаторов позволило распараллелить обращения от различных процессоров. Однако и в таком подходе неприемлемо большими казались накладные расходы для систем более чем с 32-мя процессорами.

Современные системы SMP архитектуры состоят, как правило, из нескольких однородных серийно выпускаемых микропроцессоров и массива общей памяти, подключение к которой производится либо с помощью общей шины, либо с помощью коммутатора.
ем.

Наличие общей памяти значительно упрощает организацию взаимодействия процессоров между собой и упрощает программирование.

Однако за этой кажущейся простотой скрываются большие проблемы, рисущие системам этого типа. Все они так или иначе, связаны с оперативной памятью.

Первая проблема.

В настоящее время даже в однопроцессорных системах самым узким местом является оперативная память, скорость работы которой значительно отстала от скорости работы процессора. Для того чтобы сгладить этот разрыв, современные процессоры снабжаются скоростной буферной памятью (кэш-памятью), скорость работы которой значительно выше, чем скорость работы основной памяти.
В многопроцессорных системах, построенных на базе микропроцессоров со встроенной кэш-памятью, нарушается принцип равноправного

доступа к любой точке памяти. Данные, находящиеся в кэш-памяти некоторого процессора, недоступны для других процессоров. Это означает, что при каждой модификации копии некоторой переменной, находящейся в кэш-памяти какого-либо процессора, необходимо производить синхронную модификацию самой этой переменной, расположенной в основной памяти.

С большим или меньшим успехом эти проблемы решаются в рамках общепринятой в настоящее время архитектуры ccNUMA (cache coherent Non Uniform Memory Access).

В этой архитектуре память физически распределена, но логически общедоступна. Это, с одной стороны, позволяет работать с единым адресным пространством, а, с другой, увеличивает масштабируемость систем. Когерентность кэш-памяти поддерживается на аппаратном уровне, что не избавляет, однако, от накладных расходов на ее поддержание.

В отличие от классических SMP систем память становится трехуровневой:
• кэш-память процессора;
• локальная оперативная память;
• удаленная оперативная память

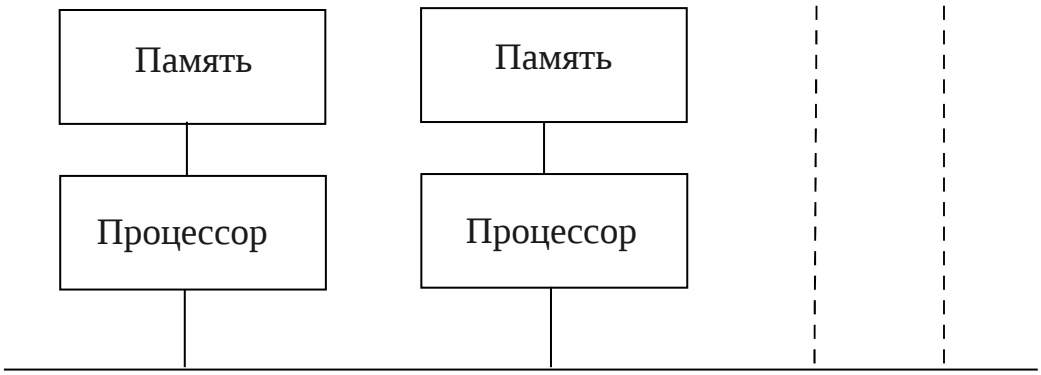
Время обращения к различным уровням может отличаться на порядок, что сильно усложняет написание эффективных параллельных программ для таких систем.
Перечисленные обстоятельства значительно ограничивают возможности по наращиванию производительности ccNUMA систем путем простого увеличения числа процессоров. Тем не менее, эта технология позволяет в настоящее время создавать системы, содержащие до 256 процессоров с общей производительностью порядка 200 Gflops. Системы этого типа серийно производятся многими компьютерными фирмами как многопроцессорные серверы с числом процессоров от 2 до 128 и прочно удерживают идерство в классе малых суперкомпьютеров. Типичными представителями данного класса суперкомпьютеров являются компьютеры SUN StarFire 15K, SGI Origin 3000, HP Superdome.

Проблема вторая

Неприятным свойством SMP систем является то, что их стоимость растет быстрее, чем производительность при увеличении числа процессоров в системе. Кроме того, из-за задержек при обращении к общей памяти неизбежно взаимное торможение при параллельном выполнении даже независимых программ.

Системы с массовым параллелизмом (MPP)

Проблемы, присущие многопроцессорным системам с общей памятью, простым и естественным образом устраняются в системах с массовым параллелизмом. Компьютеры этого типа представляют собой многопроцессорные системы с распределенной памятью, в которых с помощью некоторой коммуникационной среды объединяются однородные вычислительные узлы.



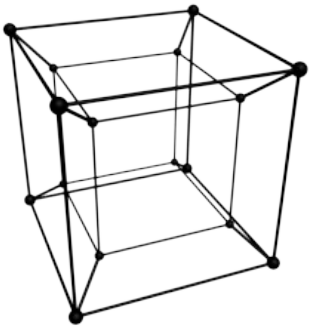
Каждый из узлов состоит из одного или нескольких процессоров, собственной оперативной памяти, коммуникационного оборудования, подсистемы ввода/вывода, т.е. обладает всем необходимым для независимого функционирования. При этом на каждом узле может функционировать либо полноценна операционная система (как в системе RS/6000 SP2), либо урезанный вариант, оддерживающий только базовые функции ядра, а полноценная ОС работает на специальном управляющем компьютере (как в системах Cray T3E, nCUBE2).

Процессоры в таких системах имеют прямой доступ тольколь к собсвенной памяти. Доступ к памяти других узлов реализуется обычно с помощью механизма передачи сообщений. Такая архитектура вычислительной системы устраняет одновременно к проблему конфликтов при обращении памяти, так и проблему когерентности кэш-памяти. Это дает возможность практически неограниченного наращивания числа процессоров в системе, увеличивая тем самым ее производительность. Успешно функционируют MPP с сотням и тысячами процессоров (ASCI White – 8192, Blue Mountain – 6144). Производительность наиболее мощных систем достигает 10 триллионов оп/сек (10 Tflops). Важным свойством MPP систем является их высокая степень масштабируемости. В зависимости от вычислительных потребностей для достижения необходимой производительности требуется просто собрать систему с нужным числом узлов.
На практике все, конечно, гораздо сложнее. Устранение одних проблем, как это обычно бывает, порождает другие. Для MPP систем на первый план выходит проблема эффективности коммуникационной среды.

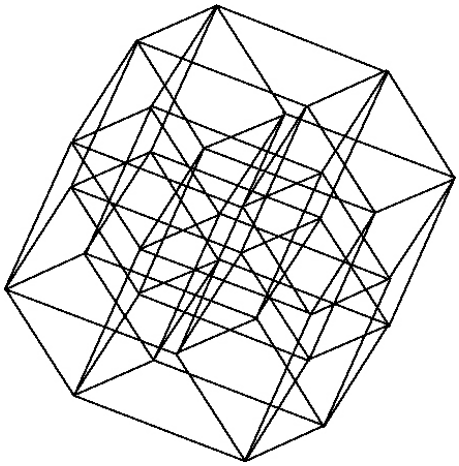
Легко сказать: “Давайте соберем систему из 1000 узлов”. Но каким образом соединить в единое целое такое множество узлов? Самым простым и наиболее эффективным было бы соединение каждого процессора с каждым. Но тогда на каждом узле бы потребовалось 999

коммуникационных каналов, желательно двунаправленных. Очевидно, что это нереально. Различные производители MPP систем использовали разные топологии. В компьютерах Intel Paragon процессоры образовывали прямоугольную двумерную сетку. Для этого в каждом узле достаточно четырех коммуникационных каналов. В компьютерах Cray T3D/T3E использовалась топология трехмерного тора. Соответственно, в узлах этого компьютера было шесть коммуникационных каналов. Фирма nCUBE использовала в своих компьютерах топологию n-мерного гиперкуба.

Тессеракт



Пентеракт



При обмене данными между процессорами, не являющимися ближайшими соседями, происходит трансляция данных через промежуточные узлы.

Очевидно, что в узлах должны быть предусмотрены какие-то аппаратные средства, которые освобождали бы центральный процессор от участия в трансляции данных. В последнее время для соединения вычислительных узлов чаще используется иерархическая система высокоскоростных коммутаторов как это впервые было реализовано в компьютерах IBM SP2. Такая топология дает возможность прямого обмена данными между любыми узлами, без участия в этом промежуточных узлов.

Системы с распределенной памятью идеально подходят для параллельного выполнения независимых программ, поскольку при том каждая программа выполняется на своем узле и никаким образом не влияетна выполнение других программ. Однако при разработке параллельных программ приходится учитывать более сложную, чем в SMP системах, организацию памяти. Оперативная память в MPP системах имеет 3-х уровневую структуру:

- кэш-память процессора;
- локальная оперативная память узла;
- оперативная память других узлов.

При этом отсутствует возможность прямого доступа к данным, расположенным в других узлах. Для их использования эти данные должны быть предварительно переданы в тот узел, который в данный момент в них нуждается. Это значительно усложняет программирование. Кроме того, обмены данными между узлами выполняются значительно медленнее, чем обработка данных в локальной оперативной памяти узлов. Поэтому написание эффективных параллельных программ для таких компьютеров представляет собой более сложную задачу, чем для SMP систем.

Кластерные системы

Кластер – это связанный набор полноценных компьютеров, используемый в качестве единого вычислительного ресурса.

Кластерные технологии стали логическим продолжением развития идей, заложенных в архитектуре MPP систем. Если процессорный модуль в MPP системе представляет собой законченную вычислительную систему, то следующий шаг напрашивается сам собой: почему бы в качестве таких вычислительных узлов не использовать обычные серийно выпускаемые компьютеры. Развитие коммуникационных технологий, а именно, появление высокоскоростного сетевого оборудования и специального программного обеспечения, такого как система MPI, реализующего механизм передачи сообщений над стандартными сетевыми протоколами, сделали кластерные технологии общедоступными.

Сегодня не составляет большого труда создать небольшую кластерную систему, объединив вычислительные мощности компьютеров отдельной лаборатории или учебного класса. Привлекательной чертой кластерных технологий является то, что они позволяют для достижения необходимой производительности объединять в единые вычислительные системы компьютеры самого разного типа, начиная от персональных компьютеров и заканчивая мощными суперкомпьютерами.

Широкое распространение кластерные технологии получили как средство создания систем суперкомпьютерного класса из составных частей массового производства, что значительно удешевляет стоимость вычислительной системы. В частности, одним из первых был реализован проект COCOA, в котором на базе 25 двухпроцессорных персональных компьютеров общей стоимостью порядка $100000 была создана система с производительностью, эквивалентной 48-процессорному Cray T3D стоимостью несколько миллионов долларов США.

Конечно, о полной эквивалентности этих систем говорить не приходится. Как указывалось в предыдущем разделе, производительность систем с распределенной памятью очень сильно зависит от производительности коммуникационной среды. Коммуникационную среду можно достаточно полно охарактеризовать

двумя параметрами: латентностью – временем задержки при посылке сообщения и пропускной способностью – скоростью передачи информации.

Так вот для компьютера Cray T3D эти параметры составляют соответственно 1 мкс и 480 Мб/сек, а для кластера, в котором в качестве коммуникационной среды использована сеть Fast Ethernet, 100 мкс и 10 Мб/сек. Это отчасти объясняет очень высокую стоимость суперкомпьютеров. При таких параметрах, как у рассматриваемого кластера, найдется не так много задач, которые могут эффективно решаться на достаточно большом числе процессоров.

Преимущества кластерной системы перед набором независимых компьютеров очевидны. Во-первых, разработано множество диспетчерских систем пакетной обработки заданий, позволяющих послать задание на обработку кластеру в целом, а не какому-то отдельному компьютеру. Эти диспетчерские системы автоматически распределяют задания по свободным вычислительным узлам или буферизуют их при отсутствии таковых, что позволяет обеспечить более равномерную и эффективную загрузку компьютеров.

Во-вторых, появляется возможность совместного использования вычислительных ресурсов нескольких компьютеров для решения одной задачи. Для создания кластеров обычно используются либо простые однопроцессорные персональные компьютеры, либо двух- или четырех-процессорные SMP-серверы. При этом не накладывается никаких ограничений на состав и архитектуру узлов. Каждый из узлов может функционировать под управлением своей собственной операционной системы. Чаще всего используются стандартные ОС: Linux, FreeBSD, Solaris, Tru64 Unix, Windows NT. В тех случаях, когда узлы кластера неоднородны, то говорят о гетерогенных кластерах.

При создании кластеров можно выделить два подхода.

Первый подход применяется при создании небольших кластерных систем. В кластер объединяются полнофункциональные компьютеры, которые продолжают работать и как самостоятельные единицы, например, компьютеры учебного класса или рабочие станции лаборатории.

Второй подход применяется в тех случаях, когда целенаправленно создается мощный вычислительный ресурс. Тогда системные блоки компьютеров компактно размещаются в специальных стойках, а для управления системой и для запуска задач выделяется один или несколько полнофункциональных компьютеров, называемых хост-компьютерами. В этом случае нет необходимости снабжать компьютеры вычислительных узлов графическими картами, мониторами, дисковыми накопителями и другим периферийным оборудованием, что значительно удешевляет стоимость системы.

Разработано множество технологий соединения компьютеров в кластер. Наиболее широко в данное время используется технология Fast Ethernet. Это обусловлено простотой ее использования и низкой стоимостью коммуникационного оборудования. Однако за это приходится расплачиваться заведомо недостаточной скоростью обменов.

Частично это положение может поправить переход на технологии Gigabit Ethernet.

Ряд фирм предлагают специализированные кластерные решения на основе более скоростных сетей, таких как SCI фирмы Scali Computer (~100 Мб/сек) и Mirynet (~120 Мб/сек). Активно включились в поддержку кластерных технологий и фирмы-производители высокопроизводительных рабочих станций (SUN, HP, Silicon Graphics).

Классификация Флинна

Самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М. Флинном. Классификация базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD,MISD,SIMD,MIMD.

SISD (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины, или иначе, машины фон-неймановского типа, например, PDP-11 или VAX 11/780. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним потоком данных. Не имеет значения тот факт, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка - как машина CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными попадают в этот класс.

SIMD (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными - элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производится либо процессорной матрицей, как в ILLIAC IV, либо с помощью конвейера, как, например, в машине CRAY-1.

MISD (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе. Ряд исследователей относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе. Будем считать, что пока данный класс пуст.

MIMD (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

В **SISD** входят однопроцессорные последовательные компьютеры типа VAX 11/780. Однако, многими критиками подмечено, что в этот класс можно включить и векторно-конвейерные машины, если рассматривать вектор как одно неделимое данное для соответствующей команды. В таком случае в этот класс попадут и такие системы, как CRAY-1, CYBER 205, машины семейства FACOM VP и многие другие.

Бесспорными представителями класса **SIMD** считаются матрицы процессоров: ILLIAC IV, ICL DAP, Goodyear Aerospace MPP, Connection Machine 1 и т.п. В таких системах единое управляющее устройство контролирует множество процессорных элементов. Каждый процессорный элемент получает от устройства управления в каждый фиксированный момент времени одинаковую команду и выполняет ее над своими локальными данными. Для классических процессорных матриц никаких вопросов не возникает, однако в этот же класс можно включить и векторно-конвейерные машины, например, CRAY-1. В этом случае каждый элемент вектора надо рассматривать как отдельный элемент потока данных.

Класс **MIMD** чрезвычайно широк, поскольку включает в себя всевозможные мультипроцессорные системы: Cm*, C.mmp, CRAY Y-MP, Denelcor HEP,BBN Butterfly, Intel Paragon, CRAY T3D и многие другие. Интересно то, что если конвейерную обработку рассматривать как выполнение множества команд (операций ступеней конвейера) не над одиночным векторным потоком данных, а над множественным скалярным потоком, то все рассмотренные выше векторно-конвейерные компьютеры можно расположить и в данном классе.

Предложенная схема классификации вплоть до настоящего времени является самой применяемой при начальной характеристике того или иного компьютера. Если говорится, что компьютер принадлежит классу SIMD или MIMD, то сразу становится понятным базовый принцип его работы, и в некоторых случаях этого бывает достаточно. Однако видны и

явные недостатки. В частности, некоторые заслуживающие внимания архитектуры, например dataflow и векторно-конвейерные машины, четко не вписываются в данную классификацию. Другой недостаток - это чрезмерная заполненность класса MIMD. Необходимо средство, более избирательно систематизирующее архитектуры, которые по Флинну попадают в один класс, но совершенно различны по числу процессоров, природе и топологии связи между ними, по способу организации памяти и, конечно же, по технологии программирования.

Наличие пустого класса (**MISD**) не стоит считать недостатком схемы. Такие классы, по мнению некоторых исследователей в области классификации архитектур, могут стать чрезвычайно полезными для разработки принципиально новых концепций в теории и практике построения вычислительных систем.

Классификация Ванга-Бриггса

В книге К.Ванга и Ф.Бриггса сделаны некоторые дополнения к классификации Флинна. Оставляя четыре ранее введенных базовых класса (SISD, SIMD, MISD, MIMD), авторы внесли следующие изменения.

Класс **SISD** разбивается на два подкласса:

архитектуры с единственным функциональным устройством, например, PDP-11;
архитектуры, имеющие в своем составе несколько функциональных устройств - CDC 6600, CRAY-1, FPS AP-120B, CDC Cyber 205, FACOM VP-200.

В класс **SIMD** также вводится два подкласса:

архитектуры с пословно-последовательной обработкой информации - ILLIAC IV, PEPE, BSP; архитектуры с разрядно-последовательной обработкой - STARAN, ICL DAP.

В классе **MIMD** авторы различают вычислительные системы со слабой связью между процессорами, к которым они относят все системы с распределенной памятью, например, Cosmic Cube,
и вычислительные системы с сильной связью (системы с общей памятью), куда попадают такие компьютеры, как C.mmp, BBN Butterfly, CRAY Y-MP, Denelcor HEP.

Параллелльное программирование

Парадигмы программирования

В настоящее время существуют два основных подхода к программированию вычислений, эти подходы в литературе часто называют *парадигмами* программирования. Прежде чем обсудить эти подходы, сделаем несколько замечаний общего характера. Развитие фундаментальных и прикладных наук, технологий требует применения все более мощных и эффективных методов и средств обработки информации. В качестве примера можно привести разработку реалистических математических моделей, которые часто оказываются настолько сложными, что не допускают точного аналитического их исследования. Единственная возможность исследования таких моделей, их верификации (то есть подтверждения правильности) и использования для прогноза - компьютерное моделирование, применение методов численного анализа. Другая важная проблема - обработка больших объемов информации в режиме реального времени. Все эти проблемы могут быть решены лишь на достаточно мощной аппаратной базе, с применением эффективных методов программирования.

Традиционная архитектура ЭВМ была последовательной. Это означало, что в любой момент времени выполнялась только одна операция и только над одним операндом. Совокупность приемов программирования, структур данных, отвечающих последовательной архитектуре компьютера, называется моделью последовательного программирования. Ее основными чертами являются применение стандартных языков программирования (как правило, это ФОРТРАН-77, ФОРТРАН-90, C/C++), достаточно простая переносимость программ с одного компьютера на другой и невысокая производительность.

Появление в середине шестидесятых первого компьютера класса суперЭВМ, разработанного в фирме CDC знаменитым Сеймуром Крэем, ознаменовало рождение новой - векторной архитектуры. Начиная с этого момента суперкомпьютером принято называть высокопроизводительный векторный компьютер. Основная идея, положенная в основу новой архитектуры, заключалась в распараллеливании процесса обработки данных, когда одна и та же операция применяется одновременно к массиву (вектору) значений. В этом случае можно надеяться на определенный выигрыш в скорости вычислений (см. главу 3 настоящего учебника). Идея параллелизма оказалась плодотворной и нашла воплощение на разных уровнях функционирования компьютера. Более подробное обсуждение аппаратной реализации параллельной обработки информации можно найти во второй главе, здесь же упомянем конвейерную обработку, многопроцессорность и т.д.

Основными особенностями модели параллельного программирования являются высокая эффективность программ, применение специальных приемов программирования и, как следствие, более высокая трудоемкость программирования, проблемы с переносимостью программ.

В настоящее время существуют два основных подхода к распараллеливанию вычислений. Это параллелизм данных и параллелизм задач. В англоязычной литературе соответствующие термины - data parallel и message passing. В основе обоих подходов лежит распределение вычислительной работы по доступным пользователю процессорам параллельного компьютера.

При этом приходится решать разнообразные проблемы. Прежде всего это достаточно равномерная загрузка процессоров, так как если основная вычислительная работа будет ложиться на один из процессоров, мы приходим к случаю обычных последовательных вычислений и в этом случае никакого выигрыша за счет распараллеливания задачи не будет. Сбалансированная работа процессоров - это первая проблема, которую следует решить при организации параллельных вычислений.

Другая и не менее важная проблема - скорость обмена информацией между процессорами. Если вычисления выполняются на высокопроизводительных процессорах, загрузка которых достаточно равномерная, но скорость обмена данными низкая, основная часть времени будет тратиться впустую на ожидание информации, необходимой для дальнейшей работы данного процессора. Рассматриваемые парадигмы программирования различаются методами решения этих двух основных проблем. Разберем более подробно параллелизм данных и параллелизм задач.

Основная идея подхода, основанного на параллелизме данных, заключается в том, что одна операция выполняется сразу над всеми элементами массива данных. Различные фрагменты такого массива обрабатываются на векторном процессоре или на разных процессорах параллельной машины. Распределением данных между процессорами занимается программа. Векторизация или распараллеливание в этом случае чаще всего выполняется уже на этапе компиляции - перевода исходного текста программы в машинные команды. Роль программиста в этом случае обычно сводится к заданию опций векторной или параллельной оптимизации компилятору, директив параллельной компиляции, использованию специализированных языков для параллельных вычислений. Наиболее распространенными языками для параллельных вычислений являются Высокопроизводительный ФОРТРАН (High Performance FORTRAN) и параллельные версии языка С (это, например, С*).

Более детальное описание рассматриваемого подхода к распараллеливанию содержит указание на следующие его основные особенности:

- Обработкой данных управляет одна программа;
- Пространство имен является глобальным, то есть для программиста существует одна единственная память, а детали структуры данных,

доступа к памяти и межпроцессорного обмена данными от него скрыты;
- Слабая синхронизация вычислений на параллельных процессорах, то есть выполнение команд на разных процессорах происходит, как правило, независимо и только лишь иногда производится согласование выполнения циклов или других программных конструкций - их синхронизация. Каждый процессор выполняет один и тот же фрагмент программы, но нет гарантии, что в заданный момент времени на всех процессорах выполняется одна и та же машинная команда;
- Параллельные операции над элементами массива выполняются одновременно на всех доступных данной программе процессорах.

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

Подход, основанный на параллелизме данных, базируется на использовании при разработке программ базового набора операций:
- операции управления данными;
- операции над массивами в целом и их фрагментами;
- условные операции;
- операции приведения;
- операции сдвига;
- операции сканирования;
- операции, связанные с пересылкой данных.

Рассмотрим эти базовые наборы операций.

Управление данными.

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

Операции над массивами

Аргументами таких операций являются массивы в целом или их фрагменты (сечения), при этом данная операция применяется одновременно (параллельно) ко всем элементам массива. Примерами операций такого типа являются вычисление поэлементной суммы массивов, умножение элементов массива на скалярный или векторный множитель и т.д. Операции могут быть и более сложными - вычисление функций от массива, например.

Условные операции

Эти операции могут выполняться лишь над теми элементами массива, которые удовлетворяют какому-то определенному условию. В сеточных методах это может быть четный или нечетный номер строки (столбца) сетки или неравенство нулю элементов матрицы.

Операции приведения

Операции приведения применяются ко всем элементам массива (или его сечения), а результатом является одно единственное значение, например, сумма элементов массива или максимальное значение его элементов.

Операции сдвига

Для эффективной реализации некоторых параллельных алгоритмов требуются операции сдвига массивов. Примерами служат алгоритмы обработки изображений, конечно-разностные алгоритмы и некоторые другие.

Операции сканирования

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

Операции пересылки данных

Это, например, операции пересылки данных между массивами разной формы (то есть имеющими разную размерность и разную протяженность по каждому измерению) и некоторые другие.

При программировании на основе параллелизма данных часто используются специализированные языки - CM FORTRAN, C*, FORTRAN+, MPP FORTRAN, Vienna FORTRAN, а также HIGH PERFORMANCE FORTRAN (HPF). HPF основан на языке программирования ФОРТРАН 90, что связано с наличием в последнем удобных операций над массивами.

Основной характеристикой при классификации многопроцессорных систем является наличие общей (SMP системы) или распределенной (MPP системы) памяти. Это различие является важнейшим фактором,

определяющим способы параллельного программирования и, соответственно, структуру программного обеспечения.

Программирование в системах с общей памятью

К системам этого типа относятся компьютеры с SMP архитектурой, различные разновидности NUMA систем и мультипроцессорные векторно-конвейерные компьютеры. Характерным словом для этих компьютеров является “единый”: единая оперативная память, единая операционная система, единая подсистема ввода-вывода. Только процессоры образуют множество. Единая UNIX-подобная операционная система, управляющая работой всего компьютера, функционирует в виде множества процессов.

Каждая пользовательская программа также запускается как отдельный процесс. Операционная система сама каким-то образом распределяет процессы по процессорам. В принципе, для распараллеливания программ можно использовать механизм порождения процессов. Однако этот механизм не очень удобен, поскольку каждый процесс функционирует в своем адресном пространстве, и основное достоинство этих систем – общая память – не может быть использован простым и естественным образом.

Для распараллеливания программ используется механизм порождения нитей (threads) – легковесных процессов, для которых не создается отдельного адресного пространства, но которые на многопроцессорных системах также распределяются по процессорам. В языке программирования C возможно прямое использование этого механизма для распараллеливания программ посредством вызова соответствующих системных функций, а в компиляторах с языка FORTRAN этот механизм используется либо для автоматического распараллеливания, либо в режиме задания распараллеливающих директив компилятору (такой подход поддерживают и компиляторы с языка C).

Все производители симметричных мультипроцессорных систем в той или иной мере поддерживают стандарт POSIX Pthread и включают в программное обеспечение распараллеливающие компиляторы для популярных языков программирования или предоставляют набор директив компилятору для распараллеливания программ. В частности, многие поставщики компьютеров SMP архитектуры (Sun, HP, SGI) в своих компиляторах предоставляют специальные директивы для распараллеливания циклов. Однако эти наборы директив, во-первых, весьма ограничены и, во-вторых, несовместимы между собой. В результате этого разработчикам приходится распараллеливать прикладные программы отдельно для каждой платформы.

Пример программы, использующей механизм нитей. Библиотека нитей – pth. Программа состоит из трех нитей – основной и двух порожденных.

Пока основная нить спит, порожденные переключаются между собой через 5 шагов цикла, выводя номер нити на каждом шаге.

Используются функции

- pth_attr_new, pth_attr_set – манипуляция атрибутами нитей;
- pth_spawn – создание нитей;
- pth_yield – переключение нитей.

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <errno.h>
        #include <sys/types.h>
        #include <sys/socket.h>
        #include <netinet/in.h>
        #include <arpa/inet.h>
        #include <signal.h>
        #include <netdb.h>
        #include <unistd.h>
        #include <pth.h>
        static void *handler(void *n)
        {
         int nn = (int) n;
         int i;
         for(i=0;;i++)
         {
          printf("%d\n", nn);
          if( i % 5 == 0 ) pth_yield(NULL);
         }
        }

        main()
       {
        pth_attr_t attr;
        pth_init();
        attr = pth_attr_new();
        pth_attr_set(attr, PTH_ATTR_NAME, "handler");
        pth_attr_set(attr, PTH_ATTR_CANCEL_STATE,
PTH_CANCEL_ASYNCHRONOUS);

        pth_spawn(attr, handler, (void *)22);
        pth_spawn(attr, handler, (void *)33);
        printf("nt=%ld\n", pth_ctrl(PTH_CTRL_GETTHREADS));
        pth_sleep(10);
}
```

Обратите внимание, что для засыпания основной нити используется функция pth_sleep, а не просто sleep. Поскольку вторая вызывает засыпание всего процесса целиком (со всеми порожденными нитями). Многие функции ввода/вывода имеют свою пару в библиотеке pth, которые имеют соответствующий префикс. Они отличаются от своих оригиналов тем, что засыпает только нить (а не весь процесс), вызвавшая их.

В последние годы все более популярной становится система программирования OpenMP, являющаяся во многом обобщением и

расширением этих наборов директив. Интерфейс OpenMP задуман как стандарт для программирования в модели общей памяти. В OpenMP входят спецификации набора директив компилятору, процедур и переменных среды. По сути дела, он реализует идею "инкрементального распараллеливания", позаимствованную из языка HPF (High Performance Fortran – Fortran для высокопроизводительных вычислений). Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP-директивы.

При этом система программирования OpenMP предоставляет разработчику большие возможности по контролю над поведением параллельного приложения. Вся программа разбивается на последовательные и параллельные области. Все последовательные области выполняет главная нить, порождаемая при запуске программы, а при входе в параллельную область главная нить порождает дополнительные нити. Предполагается, что OpenMP-программа без какой-либо модификации должна работать как на многопроцессорных системах, так и на однопроцессорных. В последнем случае директивы OpenMP просто игнорируются. Следует отметить, что наличие общей памяти не препятствует использованию технологий программирования, разработанных для систем с распределенной памятью. Многие производители SMP систем предоставляют также такие технологии программирования, как MPI и PVM. В этом случае в качестве коммуникационной среды выступает разделяемая память.

OpenMP

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP,ccNUMA, etc.) в модели общей памяти (shared memory model). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

Примерами систем с общей памятью, масштабируемых до большого числа процессоров, могут служить суперкомпьютеры Cray Origin2000 (до 128 процессоров), HP 9000 V-class (до 32 процессоров в одном узле, а в конфигурации из 4 узлов - до 128 процессоров), Sun Starfire (до 64 процессоров).

Разработкой стандарта занимается организация OpenMP ARB (ARchitecture Board), в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения. Спецификации для языков Fortran и C/C++ появились соответственно в октябре 1997 года и октябре 1998 года. Основной источник информации - сервер www.openmp.org. На сервере доступны спецификации, статьи, учебные материалы, ссылки.

Наиболее гибким, переносимым и общепринятым интерфейсом параллельного программирования является MPI (интерфейс передачи сообщений). Однако модель передачи сообщений 1) недостаточно

эффективна на SMP-системах; 2) относительно сложна в освоении, так как требует мышления в "невычислительных" терминах.

Проект стандарта X3H5 провалился, так как был предложен во время всеобщего интереса к MPP-системам, а также из-за того, что в нем поддерживается только параллелизм на уровне циклов. OpenMP развивает многие идеи X3H5.

POSIX-интерфейс для организации нитей (Pthreads) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического параллельного программирования:

- нет поддержки Fortran-а,
- слишком низкий уровень,
- нет поддержки параллелизма по данным,
- механизм нитей изначально разрабатывался не для целей организации параллелизма.

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей).
Многие поставщики SMP-архитектур (Sun,HP,SGI) в своих компиляторах поддерживают спецдирективы для распараллеливания циклов. Однако эти наборы директив, как правило, 1) весьма ограничены; 2) несовместимы между собой; в результате чего разработчикам приходится распараллеливать приложение отдельно для каждой платформы. OpenMP является во многом обобщением и расширением упомянутых наборов директив.

1. За счет идеи "инкрементального распараллеливания" OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.

2. При этом, OpenMP - достаточно гибкий механизм, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения.

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

4. Одним из достоинств OpenMP его разработчики считают поддержку так называемых "orphan" (оторванных) директив, то есть директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области.

Модель OpenMP

Согласно терминологии POSIX threads, любой UNIX-процесс состоит несколько нитей управления, которые имеют общее адресное пространство, но разные потоки команд и раздельные стэки. В простейшем случае, процесс состоит из одной нити. Нити иногда называют также потоками, легковесными процессами, LWP (light-weight processes).

В OpenMP используется терминология и модель программирования, близкая к Pthreads (динамически порождаемые нити, общие и разделяемые данные, механизм "замков" для синхронизации). Предполагается наиболее вероятным, что OpenMP будет реализован на базе Pthreads.

Директивы

Директивы OpenMP с точки зрения Фортрана являются комментариями и начинаются с комбинации символов "!$OMP". Директивы можно разделить на 3 категории: определение параллельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов - клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

Формат директив для C:
#pragma omp directive-name [clause[[,] clause]...] new-line

определение параллельной секции

#pragma omp parallel [clause[[,]clause] ...] new-line
structured-block

- private(*list*)
- firstprivate(*list*)
- default(shared | none)
- shared(*list*)
- copyin(*list*)
- reduction(*operator*: *list*)
- num_threads(*integer-expression*)

Пример программы

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

main ()
{
int size, rank;

/* Создание множества параллельных процессов и в каждом из них
задаются свои приватные переменные size и rank */

#pragma omp parallel private(size, rank)

  {

/* Каждый процесс находит свой порядковый номер и выводит его на
экран */
  rank = omp_get_thread_num();
  printf("Hello World from thread = %d\n", rank);

/* Главный процесс - master выводит на экран количество процессов */
  if (rank == 0)
    {
    size = omp_get_num_threads();
    printf("Number of threads = %d\n", size);
    }
  }  /* Завершение параллельной части */
}
```

разделение работы

циклы

```
#pragma omp for [clause[[,] clause] ... ] new-line
for-loop

private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered
schedule(kind[, chunk_size])
nowait

for-loop:
for (init-expr; var relational-op b; incr-expr) statement

init-expr:
```

var = lb
integer-type var = lb
incr-expr:
++var
var++
--var
var--
var += incr
var -= incr
var = var + incr
var = incr + var
var = var - incr

relational-op:
<
<=
>
>=

Пример программы

```
#pragma omp parallel for private(i)
#pragma omp shared(x, y, n) reduction(+: a, b)
for (i=0; i<n; i++)
{
    a = a + x[i];
    b = b + y[i];
}
```

секции

#pragma omp sections [clause[[,] clause] ...] new-line
{
[#pragma omp section new-line]
structured-block
[#pragma omp section new-line
structured-block]
...
}
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait

одиночные конструкции

#pragma omp single [clause[[,] clause] ...] new-line
structured-block

- private(list)
- firstprivate(list)
- copyprivate(list)
- nowait

```
#include <stdio.h>
void work1() {}
void work2() {}
void a10()
{
#pragma omp parallel
{
#pragma omp single
printf("Beginning work1.\n");
work1();
#pragma omp single
printf("Finishing work1.\n");
#pragma omp single nowait
printf("Finished work1 and beginning work2.\n");
work2();
}
}
```

синхронизация

#pragma omp master new-line
structured-block
Директива указывает, что блок будет выполняться нулевой нитью.

#pragma omp critical [(name)] new-line
structured-block
Блок будет выполняться только одной нитью

Кроме этих существуют директивы позволяющее выполнять

- действия по изменению значений переменных атомарно
- “скидывать” значения переменных в общую память
- определять порядок выполнения секций

Классы переменных

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

SHARED (общие; под именем A все нити видят одну переменную) и PRIVATE (приватные; под именем A каждая нить видит свою переменную).
Отдельные правила определяют поведение переменных при входе и выходе из параллельной области или параллельного цикла: REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN.

По умолчанию, все COMMON-блоки, а также переменные, порожденные вне параллельной области, при входе в эту область остаются общими (SHARED). Исключение составляют переменные - счетчики итераций в цикле, по очевидным причинам. Переменные, порожденные внутри параллельной области, являются приватными (PRIVATE). Явно назначить класс переменных по умолчанию можно с помощью клаузы DEFAULT.

SHARED
Применяется к переменным, которые необходимо сделать общими.

PRIVATE
Применяется к переменным, которые необходимо сделать приватными. При входе в параллельную область для каждой нити создается отдельный экземпляр переменной, который не имеет никакой связи с оригинальной переменной вне параллельной области.

THREADPRIVATE
Применяется к COMMON-блокам, которые необходимо сделать приватными. Директива должна применяться после каждой декларации COMMON-блока.

FIRSTPRIVATE
Приватные копии переменной при входе в параллельную область инициализируются значением оригинальной переменной.

LASTPRIVATE
По окончании параллельно цикла или блока параллельных секций, нить, которая выполнила последнюю итерацию цикла или последнюю секцию блока, обновляет значение оригинальной переменной.

REDUCTION(+:A)
Обозначает переменную, с которой в цикле производится reduction-операция (например, суммирование). При выходе из цикла, данная операция производится над копиями переменной во всех нитях, и результат присваивается оригинальной переменной.

COPYIN

Применяется к COMMON-блокам, которые помечены как THREADPRIVATE. При входе в параллельную область приватные копии этих данных инициализируются оригинальными значениями.

Runtime-процедуры

В целях создания переносимой среды запуска параллельных программ, в OpenMP определен ряд переменных среды, контролирующих поведение приложения.

В OpenMP предусмотрен также набор библиотечных процедур, которые позволяют:

во время исполнения контролировать и запрашивать различные параметры, определяющие поведение приложения (такие как число нитей и процессоров, возможность вложенного параллелизма); процедуры назначения параметров имеют приоритет над соотвествующими переменными среды.

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

OMP_GET_MAX_THREADS
Возвращает максимальное число нитей.

OMP_GET_NUM_THREADS
Возвращает фактическое число нитей в параллельной области программы.

OMP_GET_NUM_PROCS
Возвращает число процессоров, доступных приложению.

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

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

OMP_GET_NESTED / OMP_SET_NESTED
Устанавливает/запрашивает состояние флага, разрешающего вложенный параллелизм.
использовать синхронизацию на базе замков (locks).
Переменные среды

OMP_SCHEDULE

Определяет способ распределения итераций в цикле, если в директиве DO использована клауза SCHEDULE(RUNTIME).

OMP_NUM_THREADS
Определяет число нитей для исполнения параллельных областей приложения.

OMP_DYNAMIC
Разрешает или запрещает динамическое изменение числа нитей.

OMP_NESTED
Разрешает или запрещает вложенный параллелизм.

компиляция в Linux

Для компиляции в ОС Linux необходим компилятор gcc, в который включен проект gomp. Информацию по нему можно найти на gcc.gnu.org.

Вызов компилятора:
/путь_до_gcc/gcc –o прога прога.c -I/путь_до_заголовочных_файлов/ -L/путь_до_библиотек/ -fopenmp

Здесь openmp – имя препроцессора для omp директив, также подключает библиотеки gomp и pthreads

Параллельное программирование на MPP системах

Решение на компьютере вычислительной задачи для выбранного алгоритма решения предполагает выполнение некоторого фиксированного объема арифметических операций. Ускорить решение задачи можно одним из трех способов:

1. использовать более производительную вычислительную систему с более быстрым процессором и более скоростной системной шиной;
2. оптимизировать программу, например, в плане более эффективного использования скоростной кэш-памяти;
3. распределить вычислительную работу между несколькими процессорами, т.е. перейти на параллельные технологии.

Разработка параллельных программ является весьма трудоемким процессом, особенно для систем MPP типа, поэтому, прежде чем приступать к этой работе, важно правильно оценить как ожидаемый эффект от распараллеливания, так и трудоемкость выполнения этой работы. Очевидно, что без распараллеливания не обойтись при программировании алгоритмов решения тех задач, которые в принципе не могут быть решены на однопроцессорных системах. Это может проявиться в двух случаях: либо когда для решения задачи требуется слишком много времени, либо когда для программы недостаточно оперативной памяти на однопроцессорной системе. Для небольших

задач зачастую оказывается, что параллельная версия работает медленнее, чем однопроцессорная.

Такая ситуация наблюдается, например, при решении на nCUBE2 систем линейных алгебраических уравнений, содержащих менее 100 неизвестных. Заметный эффект от распараллеливания начинает наблюдаться при решении систем с 1000 и более неизвестными. На кластерных системах ситуация еще хуже.

Разработчики пакета ScaLAPACK для многопроцессорных систем с приемлемым соотношением между производительностью узла и скоростью обмена дают следующую формулу для количества процессоров, которое рекомендуется использовать при решении задач линейной алгебры:

P=(MxN)/1000000
, где M × N – размерность матрицы.

Или, другими словами, количество процессоров должно быть таково, чтобы на процессор приходился блок матрицы размером примерно 1000×1000. Эта формула, конечно, носит рекомендательный характер, но, тем не менее, наглядно иллюстрирует, для задач какого масштаба разрабатывался пакет ScaLAPACK.

Рост эффективности распараллеливания при увеличении размера решаемой системы уравнений объясняется очень просто: при увеличении размерности решаемой системы уравнений объем вычислительной работы растет пропорционально n3 , а объем обменов между процессорами пропорционально n2 . Это снижает относительную долю коммуникационных затрат при увеличении размерности системы уравнений. Однако, как мы увидим далее, не только коммуникационные издержки влияют на эффективность параллельного приложения.

Эффективность параллельных программ

В идеале решение задачи на P процессорах должно выполняться в P раз быстрее, чем на одном процессоре, или/и должно позволить решить задачу с объемами данных, в P раз большими. На самом деле такое ускорение практически никогда не достигается. Причина этого хорошо иллюстрируется законом Амдала:
S<=(1/(f+(1-f)/P))

где S – ускорение работы программы на P процессорах, а f – доля непараллельного кода в программе.
Эта формула справедлива и при программировании в модели общей памяти, и в модели передачи сообщений. Несколько разный смысл вкладывается в понятие доля непараллельного кода. Для SMP систем (модель общей памяти) эту долю образуют те операторы, которые выполняет только главная нить программы. Для MPP систем (механизм

передачи сообщений) непараллельная часть кода образуется за счет операторов, выполнение которых дублируется всеми процессорами. Оценить эту величину из анализа текста программы практически невозможно. Такую оценку могут дать только реальные просчеты на различном числе процессоров. Из формулы следует, что P-кратное ускорение может быть достигнуто, только когда доля непараллельного кода равна 0. Очевидно, что добиться этого практически невозможно.

Модели программирования для MPP

Параллельные технологии на MPP системах допускают две модели программирования (похожие на классификацию М. Флинна):
1. SPMD (Single Program Multiple Date) – на всех процессорах выполняются копии одной программы, обрабатывающие разные блоки данных;
2. MPMD (Multiple Program Multiple Date) – на процессорах выполняются разные программы, обрабатывающие разные данные. Второй вариант иногда называют функциональным распараллеливанием.

Такой подход, в частности, используется в системах обработки видеоинформации, когда множество квантов данных должны проходить несколько этапов обработки. В этом случае вполне оправданной будет конвейерная организация вычислений, при которой каждый этап обработки выполняется на отдельном процессоре. Однако такой подход имеет весьма ограниченное применение, поскольку организовать достаточно длинный конвейер, да еще с равномерной загрузкой всех процессоров, весьма сложно.

Наиболее распространенным режимом работы на системах с распределенной памятью является загрузка в некоторое число процессоров одной и той же копии программы. Рассмотрим вопрос, каким образом при этом можно достичь большей скорости решения задачи по сравнению с однопроцессорной системой.

Разбиение задачи

Разработка параллельной программы подразумевает разбиение задачи на P подзадач, каждая из которых решается на отдельном процессоре. Таким образом, упрощенную схему параллельной программы, использующей механизм передачи сообщений, можно представить следующим образом:

```
IF (процессор == 0)
        задача_1
IF (процессор == 1)
        задача_2

…..
result = reduce(result_task1, result_task2, …)
```

Здесь reduce формирует некий глобальный результат на основе локальных результатов, полученных на каждом процессоре. В этом случае одна и та же копия программы будет выполняться на P процессорах, но каждый процессор будет решать только свою подзадачу.

Если разбиение на подзадачи достаточно равномерное, а накладные расходы на обмены не слишком велики, то можно ожидать близкого к P коэффициента ускорения решения задачи.

Отдельного обсуждения требует понятие подзадача. В параллельном программировании это понятие имеет весьма широкий смысл. В MPMD модели под подзадачей понимается некоторый функционально выделенный фрагмент программы. В SPMD модели под подзадачей чаще понимается обработка некоторого блока данных. На практике процедура распараллеливания чаще всего применяется к циклам.

До выполнения вычислений необходимо принять решение о способе размещения этих массивов в памяти процессоров. Здесь возможны два варианта:
1. Все массивы целиком хранятся в каждом процессоре, тогда процедура распараллеливания сводится к вычислению стартового и конечного значений переменной цикла для каждого процессора. В каждом процессоре будет храниться своя копия всего массива, в которой будет модифицирована только часть элементов. В конце вычислений, возможно, потребуется сборка модифицированных частей со всех процессоров.
2. Все или часть массивов распределены по процессорам, т.е. в каждом процессоре хранится 1/P часть массива. Тогда может потребоваться алгоритм установления связи индексов локального массива в некотором процессоре с глобальными индексами всего массива, например, если значение элемента массива является некоторой функцией индекса. Если в процессе вычислений окажется, что какие-то требующиеся компоненты массива отсутствуют в данном процессоре, то потребуется их пересылка из других процессоров.

Отметим, что вопрос распределения данных по процессорам и связь этого распределения с эффективностью параллельной программы является основным вопросом параллельного программирования. Хранение копий всех массивов во всех процессорах во многих случаях уменьшает накладные расходы на пересылки данных, однако не дает выигрыша в плане объема решаемой задачи и создает сложности синхронизации копий массива при независимом изменении элементов этого массива различными процессорами. Распределение массивов по процессорам позволяет решать значительно более объемные задачи (их то как раз и имеет смысл распараллеливать), но тогда на первый план выступает проблема
минимизации пересылок данных.

4 этапа разработки параллельного алгоритма по Фостеру

1. разбиение задачи на минимальные независимые подзадачи (partitioning);
2. установление связей между подзадачами (communication);
3. объединение подзадач с целью минимизации коммуникаций (agglomeration);
4. распределение укрупненных подзадач по процессорам таким образом, чтобы обеспечить равномерную загрузку процессоров (mapping).

Эта схема, конечно, не более чем описание философии параллельного программирования, которая лишь подчеркивает отсутствие какого-либо формализованного подхода в параллельном программировании для MPP систем. Если 1-й и 2-й пункты имеют более или менее однозначное решение, то решение задач 3-го и 4-го пунктов основывается главным образом на интуиции программиста.

Свойства MPP системы для выполнения на ней параллельных программ

В заключение рассмотрим, какими свойствами должна обладать многопроцессорная система MPP типа для исполнения на ней параллельных программ. Минимально необходимый набор требуемых для этого средств удивительно мал:

1. Процессоры в системе должны иметь уникальные идентификаторы (номера).
2. Должна существовать функция идентификации процессором самого себя.
3. Должны существовать функции обмена между двумя процессорами: посылка сообщения одним процессором и прием сообщения другим процессором.

Парадигма передачи сообщений подразумевает асимметрию функций передачи и приема сообщений. Инициатива инициализации обмена принадлежит передающей стороне. Принимающий процессор может принять только то, что ему было послано. Различные реализации механизма передачи сообщений для облегчения разработки параллельных программ вводят те или иные расширения минимально необходимого набора функций.

организация MPI

MPI - message passing interface - библиотека функций, предназначенная для поддержки работы параллельных процессов в терминах передачи сообщений.

MPI-программа представляет собой набор независимых процессов, каждый из которых выполняет свою собственную программу (не обязательно одну и ту же), написанную на языке C или FORTRAN.

Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Как правило, каждый процесс выполняется в своем собственном адресном пространстве, однако допускается и режим разделения памяти.

MPI не специфицирует модель выполнения процесса – это может быть как последовательный процесс, так и многопотоковый. MPI не предоставляет никаких средств для распределения процессов по вычислительным узлам и для запуска их на исполнение. Эти функции возлагаются либо на операционную систему, либо на программиста.

В частности, на nCUBE2 используется стандартная команда xnc, а на кластерах – специальный командный файл mpirun, который предполагает, что исполнимые модули уже каким-то образом распределены по компьютерам кластера.

Стандарт MPI

Версия 1.0 стандарта MPI разработана в июне 1994. В его разработке участвовало 40 организаций. Они учредили Message Passing Interface Forum. Адрес его сайта: http://www.mpi-forum.org/.

Самая распространенная по реализации версия стандарта - 1.1. Она опубликована в 1995. Изменений относительно предыдущей версии – минимальны. В основном мы будем рассматривать именно этот стандарт.

Стандарт версии 2.0 вышел в 1997 году. Среди дополнений можно отметить появление поддержки C++, динамическое создание процессов (в MPI-1 необходимые процессы создавались сразу при запуске параллельной программы), параллельный ввод/вывод.

MPI не накладывает каких-либо ограничений на то, как процессы будут распределены по процессорам, в частности, возможен запуск MPI-программы с несколькими процессами на обычной однопроцессорной системе.

Реализация MPI

Существует несколько реализаций MPI. Одна из самых доступных – MPICH. Вторая версия этого пакета вышла только в августе 2006 и не вошла в состав стабильных дистрибутивов операционных систем. Первая версия доступна, например, в составе дистрибутива Linux Debian sarge (последний стабильный дистрибутив Debian на осень 2006 года).

Основные особенности mpich:
- полная поддержка MPI-1,
- MPMD программирование, включая возможность построения гетерогенных кластеров,

- поддержка привязки функций C++ стандарта MPI-2 из стандарта MPI-1
- поддержка различных типов оборудования, включая SMP и MPP.
- частичная поддержка MPI-2.

MPICH предоставляет средства для компиляции и запуска MPI-приложений.

В качестве примера рассмотрим следующую конфигурацию.

На нескольких копьютерах, объединенных в вычислительную сеть, в одноименных каталогах размещены экземпляры программы.

Как (предпочтительный) вариант можно использовать сетевой диск.

Все экземпляры программы запускаются с одного из компьютеров, с использованием команды системы rsh. Это стандартная команда для UNIX-систем позволяющая запускать программы на удаленных компьютерах.

Для имитации и отладки MPI-программ можно использовать и один компьютер. Для этого (под ОС Linux) необходимо задать ему несколько имен. Файл /etc/hosts:

127.0.0.1 localhost localpc

Здесь 127.0.0.1 – ip адрес локального интерфейса компьютера. localhost и localpc – имена компьютера, которые можно произвольно задать в любом количестве.

Для указания MPICH, на каких компьютерах должны выполняться экземпляры программы необходимо отредактировать файл /usr/lib/mpich/share/machines.LINUX. В нашем случае он будет состоять из двух строк:

localhost
localpc

Для компиляции необходимо запустить командный файл mpicc:

mpicc -o имя_программы имя_программы.c

Для запуска – mpirun:

mpirun -arch LINUX -np 2 имя_программы

-arch LINUX – название архитектуры согласуется с расширением файла /usr/lib/mpich/share/machines.LINUX

-np 2 – число запускаемых экземпляров программы

простые примеры программы

```
#include <mpi.h>

int main(int argc, char **argv)
{
            int rank, size;
            MPI_Init( &argc, &argv );
            MPI_Comm_rank(MPI_COMM_WORLD, &rank );
            MPI_Comm_size(MPI_COMM_WORLD, &size );
            while(1)
            {
              printf("я процесс %d из %d процессов\n", rank, size);
              sleep(1);
            }
           MPI_Finalize();
          return 0;
}
```

В примере каждый процесс определяет свой номер и число запущенных процессов.

```
#include <mpi.h>
main( argc, argv )
          int argc;
          char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0)    /* нулевой процесс */
    {
     strcpy(message,"привет");
     MPI_Send(message, strlen(message), MPI_CHAR, 1, 99,
MPI_COMM_WORLD);
    }
   else               /* другой процесс */
   {
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status
);
    printf("принято :%s:\n", message);
   }
   MPI_Finalize();
}
```

Нулевой процесс посылает сообщение “привет”, другой процесс его принимает и передает на стандартный выход.

Коммуникатор области связи

Для идентификации наборов процессов вводится понятие группы, объединяющей все или какую-то часть процессов. Каждая группа образует область связи, с которой связывается специальный объект – коммуникатор области связи. Процессы внутри группы нумеруются целым числом в диапазоне 0..groupsize-1. Все коммуникационные операции с некоторым коммуникатором будут выполняться только внутри области связи, описываемой этим коммуникатором.

При инициализации MPI создается предопределенная область связи, содержащая все процессы MPI-программы, с которой связывается предопределенный коммуникатор MPI_COMM_WORLD. В большинстве случаев на каждом процессоре запускается один отдельный процесс, и тогда термины процесс и процессор становятся синонимами, а величина groupsize становится равной NPROCS – числу процессоров, выделенных задаче.

Идентификатор сообщения

Идентификатор сообщения (msgtag) - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767.

При осуществлении пересылки необходимо указать идентификатор группы, внутри которой производится эта пересылка. Все процессы содержатся в группе с предопределенным идентификатором коммуникатора MPI_COMM_WORLD.

Виды функций

В MPI более сотни функций:

- функции инициализации и закрытия MPI-процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Все это множество функций предназначено для облегчения разработки эффективных параллельных программ. Пользователю принадлежит право самому решать, какие средства из предоставляемого арсенала

использовать, а какие нет. В принципе, любая параллельная программа может быть написана с использованием всего 6 MPI-функций.

Каждая из MPI функций характеризуется способом выполнения:
1. Локальная функция – выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций.
2. Нелокальная функция – для ее завершения требуется выполнение MPI-процедуры другим процессом.
3. Глобальная функция – процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи.
4. Блокирующая функция – возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.
5. Неблокирующая функция – возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

Классификация функций

Атрибуты сообщения - номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура MPI_Status, содержащая три поля: MPI_Source (номер процесса отправителя), MPI_Tag (идентификатор сообщения), MPI_Error (код ошибки); могут быть и добавочные поля.

Идентификатор сообщения (msgtag) - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767. Процессы объединяются в группы, могут быть вложенные группы. Внутри группы все процессы перенумерованы. С каждой группой ассоциирован свой коммуникатор. Поэтому при осуществлении пересылки необходимо указать идентификатор группы, внутри которой производится эта пересылка. Все процессы содержатся в группе с предопределенным идентификатором MPI_COMM_WORLD.

При описании процедур MPI слово OUT используется для обозначения "выходных" параметров, т.е. таких параметров, через которые процедура возвращает результаты.

Общие процедуры MPI

int MPI_Init(int* argc, char*** argv)

MPI_Init - инициализация параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза, а если MPI уже был инициализирован, то никакие действия не выполняются и происходит немедленный возврат из подпрограммы. Все оставшиеся MPI-процедуры могут быть вызваны только после вызова MPI_Init.

Возвращает: в случае успешного выполнения - MPI_SUCCESS, иначе - код ошибки. (То же самое возвращают и все остальные функции)

int MPI_Finalize(void)

MPI_Finalize - завершение параллельной части приложения. Все последующие обращения к любым MPI-процедурам, в том числе к MPI_Init, запрещены. К моменту вызова MPI_Finalize некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.
Сложный тип аргументов MPI_Init предусмотрен для того, чтобы передавать всем процессам аргументы main:

```
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
        ...
    MPI_Finalize();
}
```

int MPI_Comm_size(MPI_Comm comm, int* size)

Определение общего числа параллельных процессов в группе comm.

comm - идентификатор группы
OUT size - размер группы

int MPI_Comm_rank(MPI_Comm comm, int* rank)

Определение номера процесса в группе comm. Значение, возвращаемое по адресу &rank, лежит в диапазоне от 0 до size_of_group-1.

comm - идентификатор группы
OUT rank - номер вызывающего процесса в группе comm

double MPI_Wtime(void)

Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Гарантируется, что этот момент не будет изменен за время существования процесса.

Прием/передача сообщений с блокировкой

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

buf - адрес начала буфера посылки сообщения
count - число передаваемых элементов в сообщении
datatype - тип передаваемых элементов
dest - номер процесса-получателя
msgtag - идентификатор сообщения
comm - идентификатор группы

Блокирующая посылка сообщения с идентификатором msgtag, состоящего из count элементов типа datatype, процессу с номером dest. Все элементы сообщения расположены подряд в буфере buf. Значение count может быть нулем. Тип передаваемых элементов datatype должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу dest, остается за MPI. Следует специально отметить, что возврат из подпрограммы MPI_Send не означает ни того, что сообщение уже передано процессу dest, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший MPI_Send.

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

OUT buf - адрес начала буфера приема сообщения
count - максимальное число элементов в принимаемом сообщении
datatype - тип элементов принимаемого сообщения
source - номер процесса-отправителя
msgtag - идентификатор принимаемого сообщения
comm - идентификатор группы
OUT status - параметры принятого сообщения

Прием сообщения с идентификатором msgtag от процесса source с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения count. Если число принятых элементов меньше значения count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой MPI_Probe.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере buf.

В качестве номера процесса-отправителя можно указать предопределенную константу MPI_ANY_SOURCE - признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу MPI_ANY_TAG - признак того, что подходит сообщение с любым идентификатором.

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову MPI_Recv, то первым будет принято то сообщение, которое было отправлено раньше.

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

status - параметры принятого сообщения
datatype - тип элементов принятого сообщения
OUT count - число элементов сообщения

По значению параметра status данная подпрограмма определяет число уже принятых (после обращения к MPI_Recv) или принимаемых (после обращения к MPI_Probe или MPI_Iprobe) элементов сообщения типа datatype.

int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status)

source - номер процесса-отправителя или MPI_ANY_SOURCE
msgtag - идентификатор ожидаемого сообщения или MPI_ANY_TAG
comm - идентификатор группы
OUT status - параметры обнаруженного сообщения

Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра status. Следует обратить внимание, что подпрограмма определяет только факт прихода сообщения, но реально его не принимает.

Прием/передача сообщений без блокировки

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

buf - адрес начала буфера посылки сообщения
count - число передаваемых элементов в сообщении
datatype - тип передаваемых элементов
dest - номер процесса-получателя
msgtag - идентификатор сообщения
comm - идентификатор группы
OUT request - идентификатор асинхронной передачи

Передача сообщения, аналогичная MPI_Send, однако возврат из подпрограммы происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере buf. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. того момента, когда можно переиспользовать буфер buf без опасения испортить передаваемое сообщение) можно определить с помощью параметра request и процедур MPI_Wait и MPI_Test.
Сообщение, отправленное любой из процедур MPI_Send и MPI_Isend, может быть принято любой из процедур MPI_Recv и MPI_Irecv.

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)

OUT buf - адрес начала буфера приема сообщения
count - максимальное число элементов в принимаемом сообщении
datatype - тип элементов принимаемого сообщения
source - номер процесса-отправителя
msgtag - идентификатор принимаемого сообщения
comm - идентификатор группы
OUT request - идентификатор асинхронного приема сообщения

Прием сообщения, аналогичный MPI_Recv, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере buf. Окончание процесса приема можно определить с помощью параметра request и процедур MPI_Wait и MPI_Test.

int MPI_Wait(MPI_Request *request, MPI_Status *status)

request - идентификатор асинхронного приема или передачи
OUT status - параметры сообщения

Ожидание завершения асинхронных процедур MPI_Isend или MPI_Irecv, ассоциированных с идентификатором request. В случае приема, атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра status.

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)

count - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
OUT statuses - параметры сообщений

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива statuses будет установлено в соответствующее значение.

int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status)

count - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
OUT index - номер завершенной операции обмена
OUT status - параметры сообщений

Выполнение процесса блокируется до тех пор, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если несколько операций могут быть завершены, то случайным образом выбирается одна из них. Параметр index содержит номер элемента в массиве requests, содержащего идентификатор завершенной операции.

int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

incount - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
OUT outcount - число идентификаторов завершившихся операций обмена
OUT indexes - массив номеров завершившихся операции обмена
OUT statuses - параметры завершившихся сообщений

Выполнение процесса блокируется до тех пор, пока по крайней мере одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр outcount содержит число завершенных операций, а первые outcount элементов массива indexes содержат номера элементов массива requests с их идентификаторами. Первые outcount элементов массива statuses содержат параметры завершенных операций.

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

request - идентификатор асинхронного приема или передачи
OUT flag - признак завершенности операции обмена
OUT status - параметры сообщения

Проверка завершенности асинхронных процедур MPI_Isend или MPI_Irecv, ассоциированных с идентификатором request. В параметре flag возвращает значение 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра status.

int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses)

count - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
OUT flag - признак завершенности операций обмена
OUT statuses - параметры сообщений

В параметре flag возвращает значение 1, если все операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве statuses). В противном случае возвращается 0, а элементы массива statuses неопределены.

int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)

count - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
OUT index - номер завершенной операции обмена
OUT flag - признак завершенности операции обмена
OUT status - параметры сообщения

Если к моменту вызова подпрограммы хотя бы одна из операций обмена завершилась, то в параметре flag возвращается значение 1, index содержит номер соответствующего элемента в массиве requests, a status - параметры сообщения.

int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

incount - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
OUT outcount - число идентификаторов завершившихся операций обмена
OUT indexes - массив номеров завершившихся операции обмена
OUT statuses - параметры завершившихся операций

Данная подпрограмма работает так же, как и MPI_Waitsome, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение outcount будет равно нулю.

int MPI_Iprobe(int source, int msgtag, MPI_Comm comm, int *flag, MPI_Status *status)

source - номер процесса-отправителя или MPI_ANY_SOURCE
msgtag - идентификатор ожидаемого сообщения или MPI_ANY_TAG
comm - идентификатор группы
OUT flag - признак завершенности операции обмена
OUT status - параметры обнаруженного сообщения

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре flag возвращает значение 1, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично MPI_Probe), и значение 0, если сообщения с указанными атрибутами еще нет.

Объединение запросов на взаимодействие

Процедуры данной группы позволяют снизить накладные расходы, возникающие в рамках одного процессора при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером. Несколько запросов на прием и/или передачу могут объединяться вместе для того, чтобы далее их можно было бы запустить одной командой. Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

int MPI_Send_init(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

buf - адрес начала буфера посылки сообщения
count - число передаваемых элементов в сообщении
datatype - тип передаваемых элементов
dest - номер процесса-получателя
msgtag - идентификатор сообщения
comm - идентификатор группы
OUT request - идентификатор асинхронной передачи

Формирование запроса на выполнение пересылки данных. Все параметры точно такие же, как и у подпрограммы MPI_Isend, однако в отличие от нее пересылка не начинается до вызова подпрограммы MPI_Startall.

int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)

OUT buf - адрес начала буфера приема сообщения
count - число принимаемых элементов в сообщении
datatype - тип принимаемых элементов
source - номер процесса-отправителя
msgtag - идентификатор сообщения
comm - идентификатор группы
OUT request - идентификатор асинхронного приема

Формирование запроса на выполнение приема данных. Все параметры точно такие же, как и у подпрограммы MPI_Irecv, однако в отличие от нее реальный прием не начинается до вызова подпрограммы MPI_Startall.

MPI_Startall(int count, MPI_Request *requests)

count - число запросов на взаимодействие
OUT requests - массив идентификаторов приема/передачи

Запуск всех отложенных взаимодействий, ассоциированных вызовами подпрограмм MPI_Send_init и MPI_Recv_init с элементами массива запросов requests. Все взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью процедур MPI_Wait и MPI_Test.

Совмещенные прием/передача сообщений

int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype, int dest, int stag, void *rbuf, int rcount, MPI_Datatype rtype, int source, MPI_Datatype rtag, MPI_Comm comm, MPI_Status *status)

sbuf - адрес начала буфера посылки сообщения
scount - число передаваемых элементов в сообщении
stype - тип передаваемых элементов
dest - номер процесса-получателя
stag - идентификатор посылаемого сообщения
OUT rbuf - адрес начала буфера приема сообщения
rcount - число принимаемых элементов сообщения
rtype - тип принимаемых элементов
source - номер процесса-отправителя
rtag - идентификатор принимаемого сообщения
comm - идентификатор группы
OUT status - параметры принятого сообщения

Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией MPI_Sendrecv, может быть принято обычным образом, и точно также операция MPI_Sendrecv может принять сообщение, отправленное обычной операцией MPI_Send. Буфера приема и посылки обязательно должны быть различными.

Коллективные взаимодействия процессов

В операциях коллективного взаимодействия процессов участвуют все процессы коммуникатора. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)

OUT buf - адрес начала буфера посылки сообщения
count - число передаваемых элементов в сообщении
datatype - тип передаваемых элементов
source - номер рассылающего процесса
comm - идентификатор группы

Рассылка сообщения от процесса source всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера buf процесса source будет скопировано в локальный буфер процесса. Значения параметров count, datatype и source должны быть одинаковыми у всех процессов.

int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int dest, MPI_Comm comm)

sbuf - адрес начала буфера посылки
scount - число элементов в посылаемом сообщении
stype - тип элементов отсылаемого сообщения
OUT rbuf - адрес начала буфера сборки данных
rcount - число элементов в принимаемом сообщении
rtype - тип элементов принимаемого сообщения
dest - номер процесса, на котором происходит сборка данных
comm - идентификатор группы
OUT ierror - код ошибки

Сборка данных со всех процессов в буфере rbuf процесса dest. Каждый процесс, включая dest, посылает содержимое своего буфера sbuf процессу dest. Собирающий процесс сохраняет данные в буфере rbuf, располагая их в порядке возрастания номеров процессов. Параметр rbuf имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров count, datatype и dest должны быть одинаковыми у всех процессов.

int MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

sbuf - адрес начала буфера для аргументов
OUT rbuf - адрес начала буфера для результата
count - число аргументов у каждого процесса
datatype - тип аргументов
op - идентификатор глобальной операции
comm - идентификатор группы

Выполнение count глобальных операций op с возвратом count результатов во всех процессах в буфере rbuf. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров count и datatype у всех процессов должны быть

одинаковыми. Из соображений эффективности реализации предполагается, что операция op обладает свойствами ассоциативности и коммутативности.

int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

sbuf - адрес начала буфера для аргументов
OUT rbuf - адрес начала буфера для результата
count - число аргументов у каждого процесса
datatype - тип аргументов
op - идентификатор глобальной операции
root - процесс-получатель результата
comm - идентификатор группы

Функция аналогична предыдущей, но результат будет записан в буфер rbuf только у процесса root.

Синхронизация процессов

int MPI_Barrier(MPI_Comm comm)

comm - идентификатор группы
Блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы comm также не выполнят эту процедуру.

Работа с группами процессов

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

comm - идентификатор группы
color - признак разделения на группы
key - параметр, определяющий нумерацию в новых группах
OUT newcomm - идентификатор новой группы

Данная процедура разбивает все множество процессов, входящих в группу comm, на непересекающиеся подгруппы - одну подгруппу на каждое значение параметра color (неотрицательное число). Каждая новая подгруппа содержит все процессы одного цвета. Если в качестве color указано значение MPI_UNDEFINED, то в newcomm будет возвращено значение MPI_COMM_NULL.

int MPI_Comm_free(MPI_Comm comm)

OUT comm - идентификатор группы
Уничтожает группу, ассоциированную с идентификатором comm, который после возвращения устанавливается в MPI_COMM_NULL.

Предопределенные константы

Предопределенные константы типа элементов сообщений

Константы MPI Тип в C

MPI_CHAR signed char
MPI_SHORT signed int
MPI_INT signed int
MPI_LONG signed long int
MPI_UNSIGNED_CHAR unsigned char
MPI_UNSIGNED_SHORT unsigned int
MPI_UNSIGNED unsigned int
MPI_UNSIGNED_LONG unsigned long int
MPI_FLOAT float
MPI_DOUBLE double
MPI_LONG_DOUBLE long double

Другие предопределенные типы

MPI_Status - структура; атрибуты сообщений; содержит три обязательных поля:

MPI_Source (номер процесса отправителя)
MPI_Tag (идентификатор сообщения)
MPI_Error (код ошибки)

MPI_Request - системный тип; идентификатор операции посылки-приема сообщения

MPI_Comm - системный тип; идентификатор группы (коммуникатора)

MPI_COMM_WORLD - зарезервированный идентификатор группы, состоящей их всех процессов приложения

Константы-пустышки

MPI_COMM_NULL
MPI_DATATYPE_NULL
MPI_REQUEST_NULL

Константа неопределенного значения

MPI_UNDEFINED

Глобальные операции

MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD

Любой процесс/идентификатор

MPI_ANY_SOURCE
MPI_ANY_TAG

Код успешного завершения процедуры

MPI_SUCCESS

Parallel Virtual Machine (PVM)

Это программный пакет, позволяющий объединять компьютеры в кластеры и предоставляющий возможности управления процессами с помощью механизма передачи сообщений. Существуют реализации PVM для самых различных платформ.

PVM позволяет видеть набор гетерогенных вычислительных систем как одну параллельную виртуальную машину.

Гетерогенность подразумевает различия в
-архитектуре
-формате данных
-скорости вычислений
-загрузки машин
-загрузки сети

1989 Oak Ridge National Laboratory v 1
1991 University of Tennessee v 2
1993 v 3

Принципы PVM
-конфигурируемый на пользовательском уровне набор вычислительных узлов
-прозрачный доступ к ЭВМ, входящих в набор;
-единица параллелизма в PVM - это задача:
-независимая последовательность вычислений/обменов;
-задачи обмениваются сообщениями;
-поддержка вычислений/обменов между разными типами ЭВМ, данными, сетевыми технологиями;
-использование специальных средств для обмена сообщениями внутри многопроцессорных ЭВМ;

PVM состоит из двух частей:
1 демон pvmd3

2 библиотека интерфейсных функций libgpvm3.a

Демон должен запускаеться на всех машинах пула.

В библиотеке содержится функционально полный набор примитивов для взаимодействия между задачами приложения

Пример программы, состоящей из двух частей.

```
main()
{
   int cc, tid, msgtag;
   char buf[100];

   printf("i'm t%x\n", pvm_mytid());

   cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

   if (cc == 1) {
         msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("from t%x: %s\n", tid, buf);
   } else
      printf("can't start hello_other\n");

   pvm_exit();
}
```

Запуск демона присходит автоматически при старте консоли "pvm"

С помощью консоли можно добавлять/удалять вычислительные узлы из виртуальной машины и, что самое главное, запускать процессы на выполнение.

исходный код hello_other

```
main()
{
   int ptid, msgtag;
   char buf[100];

   ptid = pvm_parent();

   strcpy(buf, "hello, world from ");
   gethostname(buf + strlen(buf), 64);
    msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);
```

```
   pvm_exit();
}
```

Запуск демона присходит автоматически при старте консоли "pvm"

С помощью консоли можно добавлять/удалять вычислительные узлы из виртуальной машины и, что самое главное, запускать процессы на выполнение.

Команды консоли pvm:

```
 add  hostname(s)    - добавить хосты к виртуальной машине
 alias               - определить/перечислить алиасы команды
 conf                - выдать конфигурацию виртуальной машины
 delete  hostname(s) - удалить хосты из виртуальной машины
 echo                - аргументы для эхо
 export              - добавить переменные обстановки для создания
                     списка экспорта
 halt                - остановить  pvmds
 help [command]      - напечатать вспомогательную информацию о
команде
 id                  - напечатать идентификатор консольной задачи
 jobs                - перечислить текущие задания
 kill  task-tid      - прекратить указанные задачи
 mstat host-tid      - выдать статусы хостов
 ps -a               - перечислить все PVM-задачи
 pstat task-tid      - выдать статусы задач
 quit                - остановить консоль
 reset               - сбросить все задачи
 setenv              - выдать/установить переменные обстановки
 sig  signum task    - передать сигнал задаче
 spawn [opt] a.out   - создать задачу
        возможны следующие opt:
       -(count)  число задач, по умолчанию 1
       -(host)   создать хост, по умолчанию произвольный
       -(ARCH)   создать хосты из  ARCH
       -?        задействовать отладку
       ->        перенаправить выход задачи на консоль
       -> file   перенаправить выход задачи в файл
       ->>file   перенаправить выход задачи в дополнение файла
 trace             - установить/показать маску слежения за событиями
 unexport - удалить переменные обстановки из списка создания экспорта
 unalias - уничтожить алиас команды
 version - показать версии из библиотеки libpvm
```

Компиляция

Для компиляции не требуется никаких дополнительных к стандартному компилятору средств. Достаточно подключить библиотеку.

Например:

cc -o hello_other hello_other.c -lpvm3

распараллеливание вычислений в PVM

В PVM существует 3 способа организации многозадачности:
1 толпа (crowd),
2 дерево,
3 гибридный.

В первом исполняются похожие или одинаковые процессы. Он делится на два вида "master-slave" и "node-only". Число процесов определяется при запуске.

Во втором способе процесы порождаются динамически. Он используется реже, чем первый. Вызванный с консоли процесс ветвится, делясь на подпроцессы, самостоятельно. Этот способ используется, когда сложно предсказать необходимое для эффективного решения задачи число процессов. Например при реализации алгоритма "ветвей и границ".

Второй способ проиллюстрирован вводным примером.

Управление процессами

pvm_spawn - создание процесса
pvm_tasks - информация о задачах выполняющихся на ВМ

pvm_exit - отключение от ВМ

pvm_mytid - определение номера задачи
pvm_parent - определение номера задачи предка

номера задач необходимы для обмена сообщениями

Обмен сообщениями

Проходит в три этапа.

1 инициализация сообщения
используются функции pvm_initsend() или pvm_mkbuf()
pvm_initsend(кодирование) используется для работы только
с одним буфером отправки

кодирование:
PvmDataDefault
PvmDataRaw - без кодирования
PvmDataInPlace - без лишнего копирования

pvm_mkbuf(кодирование)

используется реже в том случае если необходима работа
с несколькими буферами. Но активный буфер может быть
только один. Переключение между буферами
идентификаторстарогобуфера =
pvm_setsbuf(идентификаторновогобуфера)
идентификаторстарогобуфера =
pvm_setrbuf(идентификаторновогобуфера)

Определить какой сейчас активен
идентификаторбуфера = pvm_getsbuf( );
идентификаторбуфера = pvm_getrbuf( );

2 "пакетирование"
группа функций pvm_pk...()

int info = pvm_pkbyte(char *cp, int nitem, int stride)
int info = pvm_pkcplx(float *xp, int nitem, int stride)
int info = pvm_pkdcplx(double *zp, int nitem, int stride)
int info = pvm_pkdouble(double *dp, int nitem, int stride)
int info = pvm_pkfloat(float *fp, int nitem, int stride)
int info = pvm_pkint(int *np, int nitem, int stride)
int info = pvm_pklong(long *np, int nitem, int stride)
int info = pvm_pkshort(short *np, int nitem, int stride)
int info = pvm_pkstr(char *cp)

int info = pvm_packf(const char *fmt, ...) - как printf

nitem число передаваемых элементов
stride 1 - векторная запаковка 2 - поэлеменная

На принимающей стороне сообщение необходимо распаковать
pvm_upk...()

3 прием-передача
pvm_send(идентификаторзадачи, тэг) - точка-точка
pvm_mcast(массividентификаторовзадач, числозадач, тэг) - передача
нескольким
pvm_psend(идентификаторзадачи, тэг, указатель, числоэлементов, тип)
- передача массива
тип:
PVM_STR PVM_FLOAT
PVM_BYTE PVM_CPLX
PVM_SHORT PVM_DOUBLE
PVM_INT PVM_DCPLX
PVM_LONG PVM_UINT
PVM_USHORT PVM_ULONG

идентификаторбуфера = pvm_recv(идентификаторзадачи, тэг)

идентификаторбуфера = pvm_nrecv(идентификаторзадачи, тэг)
неблокирующий прием возвращает 0, если сообщение не пришло.

Другие функции для работы с сообщениями
информацию по ним можно уточнить
man имя_функции

```
pvm_probe( int tid, int msgtag )
pvm_trecv( int tid, int msgtag, struct timeval *tmout )
pvm_bufinfo( int bufid, int *bytes, int *msgtag, int *tid )
pvm_precv(  int tid, int msgtag, void *vp, int cnt,
                    int type, int *rtid, int *rtag, int *rcnt )

pvm_recvf(int (*new)(int buf, int tid, int tag))
```

Динамическая группировка процессов

Надстройка над библиотечными фукнциями
Действия выполняются не стандартным демоном pvmd,
а сервером pvmgs который стартует при вызове первой групповой функции.

номерчлена = pvm_joingroup("имя группы") - присоединиться (создать)
pvm_lvgroup("имя группы") - покинуть

pvm_barrier("имя группы", счетчик)
счетчик - число процессов, которые должны вызвать pvm_barrier, чтобы продолжилось выполнение задачи.

pvm_bcast("имя группы", тэг)

```
pvm_reduce(   функция, массив,
            числоэлементов, тип,
            тэг, "имя группы", идентификаторкорневогопроцесса
           )
```

стандартные имена для функций
PvmMax
PvmMin
PvmSum
PvmProduct

```
int tid = pvm_gettid( char *group, int inum )
int inum = pvm_getinst( char *group, int tid )
int size = pvm_gsize( char *group )
```