Ответы на вопросы по курсу «Параллельная обработка данных»

БИЛЕТ 1. Виды параллельной обработки данных, их особенности.

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность.

Параллельная обработка. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за 1000/N единиц времени. УВЕЛИЧЕНИЕ КОЛИЧЕСТВА НЕЗАВИСИМО РАБОТАЮЩИХ УСТРОЙСТВ.

Конвейерная обработка. Что необходимо для сложения двух вещественных чисел, представленных в форме с плавающей запятой? Целое множество мелких операций таких, как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.п. Процессоры первых компьютеров выполняли все эти "микрооперации" для каждой пары аргументов последовательно одна за одной до тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых. УСЛОЖНИТЬ САМО УСТРОЙСТВО, ЧТОБЫ НА РАЗНЫХ ЭТАПАХ МОГЛИ НАХОДИТЬСЯ РАЗНЫЕ ДАННЫЕ.

Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передавал бы результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если каждую микрооперацию выделить в отдельный этап (или иначе говорят - ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находится первые пять пар аргументов, а весь набор из ста пар будет обработан за 5+99=104 единицы времени - ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера). ТЕ СУЩЕСТВУЕТ НЕКОТОРАЯ ЗАДЕРЖКА ДЛЯ ТОГО ЧТОБЫ ЗАПОЛНИТЬ ВСЕ ЭТАПЫ КОНВЕЕРА, НО КОГДА ОНА ЗАПОЛНЕНА ДАЛЬШЕ ПРОИСХОДИТ УСКОРЕНИЕ ОБРАБОТКИ.

БИЛЕТ 2-3. История появления параллелизма в архитектуре ЭВМ: IBM 701, 704, 709, IBM STRETCH, ATLAS, CDC 6600, CDC 7600, ILLIAC IV.

IBM 701 (1953), **IBM 704** (1955): **разрядно-параллельная память, разрядно-параллельная арифметика**.

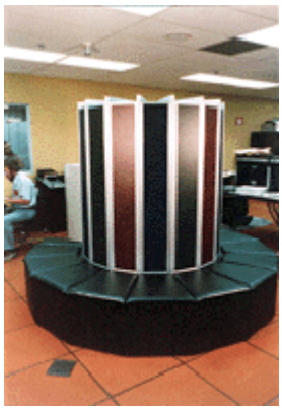
Все самые первые компьютеры (EDSAC, EDVAC, UNIVAC) имели разрядно-последовательную память, из которой слова считывались последовательно бит за битом. Первым коммерчески доступным компьютером, использующим разрядно-параллельную память (на CRT) и разрядно-параллельную арифметику, стал IBM 701, а наибольшую 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. ПЕРВЫЕ ЭВМ СЧИТЫВАЛИ СЛОВА ПО БИТАМ, И БРАБАТЫВАЛИ ИХ ТОЖ ПО БИТАМ,IBM701 IBM704 СЧИТЫВАЛИ ЗА РАЗ ВСЕ СЛОВО, И ИМЕЛИ РАЗРЯДНО ПАРАЛЛЕЛЬНУЮ АРИФМЕТИКУ.

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.
ПРИДУМАЛИ СНОБЖАТЬ УСТРОЙСТВА ВВОДА\ВЫВОДА СОБСТВЕНЫМИ ПРОЦЕССОРАМИ ДЛЯ ТОГО ЧТО БЫ ВСЯ СКОРОСТЬ СИСТЕМЫ НЕ БЫЛА ТАК СИЛЬНО ОГРАНИЧЕНА СКОРОСТЬЮ ЕЕ САМОГО МЕДЛЕННОГО УЗЛА.
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. Для сравнения с сегодняшним днем приведем некоторые параметры компьютера:
время такта 100нс,
производительность 2-3 млн. операций в секунду,
оперативная память разбита на 32 банка по 4096 60-ти разрядных слов,
цикл памяти 1мкс,
10 независимых функциональных устройств.
Машина имела громадный успех на научном рынке, активно вытесняя машины фирмы IBM.
CDC 7600 (1969): **конвейерные независимые функциональные устройства**.
CDC выпускает компьютер CDC-7600 с восемью независимыми конвейерными функциональными устройствами - сочетание параллельной и конвейерной обработки.
Основные параметры:
такт 27,5 нс,
10-15 млн. опер/сек.,
8 конвейерных ФУ,
2-х уровневая память.
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.

Иерархия памяти.
Иерархия памяти пямого отношения к параллелизму не имеет, однако, безусловно, относится к тем особенностям архитектуры компьютеров, которые имеет огромное значение для повышения их производительности (сглаживание разницы между скоростью работы процессора и временем выборки из памяти). Основные уровни: регистры, кэш-память, оперативная память, дисковая память. Время выборки по уровням памяти от дисковой памяти к регистрам уменьшается, стоимость в пересчете на 1 слово (байт) растет. В настоящее время, подобная иерархия поддерживается даже на персональных компьютерах.

БИЛЕТ 4. Оценка вычислительной сложности больших задач.

Задача 1.
Рассмотрим модель атмосферы как важнейшей составляющей климата и предположим, что мы интересуемся развитием атмосферных процессов на протяжении, например, 100 лет. При построении алгоритмов нахождения численных решений используется упоминавшийся ранее принцип дискретизации. Общее число элементов, на которые разбивается атмосфера в современных моделях, определяется сеткой с шагом в 1° по широте и долготе на всей поверхности земного шара и 40 слоями по высоте. Это дает около 2,6*10^6 элементов (360 * 180 * 40 = 2,592,000). Каждый элемент описывается примерно 10 компонентами. Следовательно, в любой фиксированный момент времени состояние атмосферы на земном шаре характеризуется ансамблем из 2,6*10^7 чисел. Условия обработки численных результатов требуют нахождения всех ансамблей через каждые 10 минут, т. е. за период 100 лет необходимо определить около 5,3*10^6 (5,256,000 = 5,2*1^6) ансамблей. Итого, только за один численный эксперимент приходится вычислять 1,4*10^14 значимых результатов промежуточных вычислений. Если теперь принять во внимание, что для получения и дальнейшей обработки каждого промежуточного результата нужно выполнить 10^2— 10^3 арифметических операций, то это означает, что для проведения одного численного эксперимента с глобальной моделью атмосферы необходимо выполнить порядка 10^{16}—10^{17} арифметических операций с плавающей запятой.
Таким образом, вычислительная система с производительностью 10^{12} операций в секунду будет осуществлять такой эксперимент при полной своей загрузке и эффективном программировании в течение нескольких часов. Использование полной климатической модели увеличивает это время, как минимум, на порядок. Еще на порядок может увеличиться время за счет не лучшего программирования и накладных расходов при компиляции программ и т. п.

Задача 2.
Рассчитывались различные варианты дозвукового обтекания летательного аппарата сложной конструкции. Математическая модель требует задания граничных условий на бесконечности. Реально область исследования берется конечной. Однако из-за обратного влияния границы ее удаление от объекта должно быть значительным по всем направлениям. На практике это составляет десятки длин размера аппарата. Таким образом, область исследования оказывается трехмерной и весьма большой. При построении алгоритмов нахождения численных решений опять используется принцип дискретизации. Из-за сложной конфигурации летательного аппарата разбиение выбирается очень неоднородным. Общее число элементов, на которые разбивается область, определяется сеткой с числом шагов порядка 10^2 по каждому измерению, т. е. всего будет порядка 10^6 элементов. В каждой точке надо знать 5 величин. Следовательно, на одном временном слое число неизвестных будет равно $5x10^6$. Для изучения нестационарного режима приходится искать решения в 10^2—10^4 слоях по времени. Поэтому одних только значимых результатов промежуточных вычислений необходимо найти около
10^9—10^{11}. Для получения каждого из них и дальнейшей его обработки нужно выполнить 10^2—10^3 арифметических операций. И это только для одного варианта компоновки и режима обтекания. А всего требуется провести расчеты для десятков вариантов. Приближенные оценки показывают, что общее число операций для решения задачи обтекания летательного аппарата в рамках современной модели составляет величину 10^{15}—10^{16}. Для достижения реального времени выполнения таких расчетов быстродействие вычислительной системы должно быть не менее 10^9—10^{10} арифметических операций с плавающей запятой в секунду при оперативной памяти не менее 10^9 слов.
Задача 3.
Пусть исследуемые объекты являются трехмерными. Чтобы получить приемлемую точность численного решения, объект нужно покрыть сеткой не менее чем 100x100x100 узлов. В каждой точке сетки нужно определить 5—20 функций. Если изучается нестационарное поведение объекта, то состояние всего ансамбля значений функций нужно определить в 10^2—10^4 моментах времени. Поэтому только значимых результатов промежуточных вычислений для подобных объектов нужно получить порядка 10^9—10^{11}. Теперь надо принять во внимание, что на вычисление и обработку каждого из промежуточных результатов, как показывает практика, требуется в среднем выполнить 10^2—10^3 арифметических операций. И вот мы уже видим, что для проведения только одного варианта численного эксперимента число операций порядка 10^{11}—10^{14} является вполне рядовым. А теперь учтем необходимое число вариантов, накладные расходы на время решения задачи, появляющиеся за счет качества программирования, компиляции и работы операционной системы.
Билет 5. Микроэлектроника и архитектура: оценка вклада в увеличение производительности компьютеров.
А почему суперкомпьютеры считают так быстро? Вариантов ответа может быть несколько, среди которых два имеют явное преимущество: развитие элементной базы и использование новых решений в архитектуре компьютеров.
Попробуем разобраться, какой из этих факторов оказывается решающим для достижения рекордной производительности. Обратимся к известным историческим фактам. На одном из первых компьютеров мира - EDSAC, появившемся в 1949 году в Кембридже и имевшем время такта 2 микросекунды (2*10-6 секунды), можно было выполнить 2*n арифметических операций за 18*n миллисекунд, то есть в среднем 100 арифметических операций в секунду. Сравним с одним вычислительным узлом современного суперкомпьютера Hewlett-Packard V2600: время такта приблизительно 1.8 наносекунды

(1.8*10-9 секунд), а пиковая производительность около 77 миллиардов арифметических операций в секунду.
Что же получается? За полвека производительность компьютеров выросла более, чем в **семьсот миллионов** раз. При этом выигрыш в быстродействии, связанный с уменьшением времени такта с 2 микросекунд до 1.8 наносекунд, составляет лишь около 1000 раз. Откуда же взялось остальное? Ответ очевиден -- использование новых решений в архитектуре компьютеров. Основное место среди них занимает принцип параллельной обработки данных, воплощающий идею одновременного (параллельного) выполнения нескольких действий.

Билет 6. Закон Амдала, его следствие, суперлинейное ускорение.

Предположим, что в вашей программе доля операций, которые нужно выполнять последовательно, равна f, где 0<=f<=1 (при этом доля понимается не по статическому числу строк кода, а по числу операций в процессе выполнения). Крайние случаи в значениях f соответствуют полностью параллельным (f=0) и полностью последовательным (f=1) программам. Так вот, для того, чтобы оценить, какое ускорение S может быть получено на компьютере из 'p' процессоров при данном значении f, можно воспользоваться законом Амдала:
S<=1/(f+(1-f)/p)
Если 9/10 программы исполняется параллельно, а 1/10 по-прежнему последовательно, то ускорения более, чем в 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0).
Посмотрим на проблему с другой стороны: а какую же часть кода надо ускорить (а значит и предварительно исследовать), чтобы получить заданное ускорение? Ответ можно найти в **следствии из закона Амдала**: для того чтобы ускорить выполнение программы в q раз необходимо ускорить не менее, чем в q раз не менее, чем (1-1/q)-ю часть программы. Следовательно, если есть желание ускорить программу в 100 раз по сравнению с ее последовательным вариантом, то необходимо получить не меньшее ускорение не менее, чем на 99.99% кода, что почти всегда составляет значительную часть программы!
Отсюда первый вывод - прежде, чем переделывать код для перехода на параллельный надо подумать. Если оценив заложенный в программе алгоритм вы поняли, что доля последовательных операций велика, то на значительное ускорение рассчитывать не приходится и нужно думать о замене отдельных компонент алгоритма.
В ряде случаев последовательный характер алгоритма изменить не так сложно. Допустим, что в программе есть следующий фрагмент для вычисления суммы n чисел:

```
s = 0
Do i = 1, n
  s = s + a(i)
EndDo
```

(можно то же самое на любом другом языке)
По своей природе он строго последователен, так как на i-й итерации цикла требуется результат с (i-1)-й и все итерации выполняются одна за другой. Имеем 100% последовательных операций, а значит и никакого эффекта от использования параллельных компьютеров. Вместе с тем, выход очевиден. Поскольку в большинстве реальных программ (вопрос: а почему в большинстве, а не во всех?) нет существенной разницы, в каком порядке складывать числа, выберем иную схему сложения. Сначала найдем сумму пар соседних элементов: a(1)+a(2), a(3)+a(4), a(5)+a(6) и т.д. Заметим, что при такой схеме все пары можно складывать одновременно! На следующих шагах будем действовать абсолютно аналогично, получив вариант параллельного алгоритма.
Казалось бы в данном случае все проблемы удалось разрешить. Но представьте, что доступные вам процессоры разнородны по своей производительности. Значит будет такой

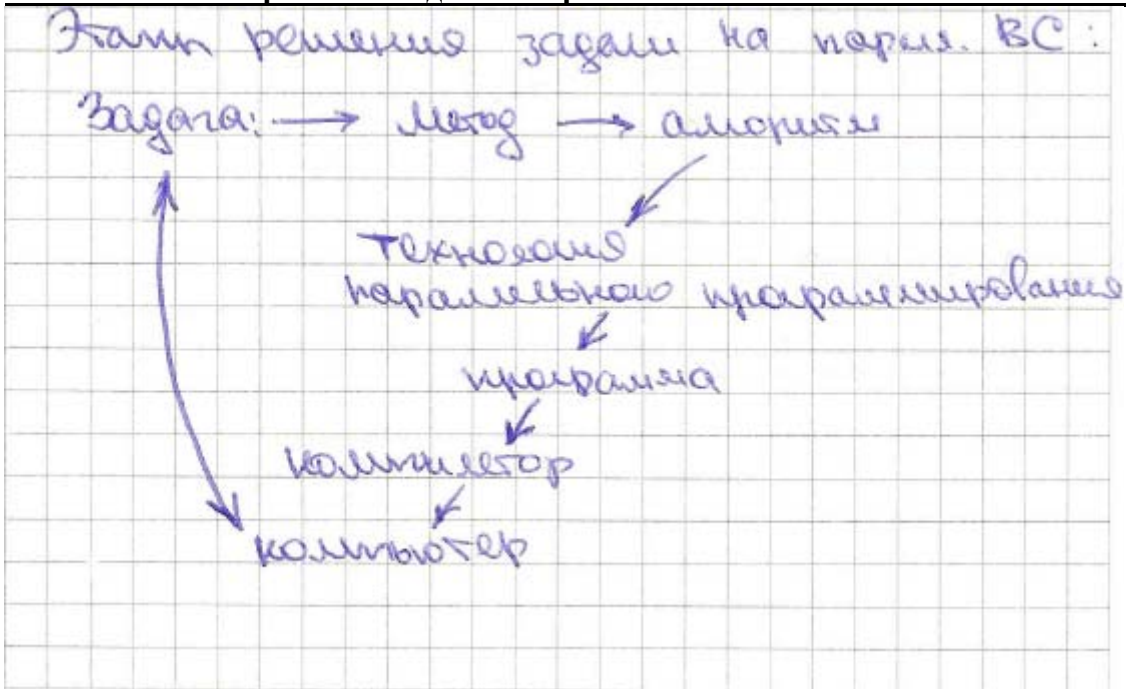
момент, когда кто-то из них еще трудится, а кто-то уже все сделал и бесполезно простаивает в ожидании. Если разброс в производительности компьютеров большой, то и эффективность всей системы при равномерной загрузке процессоров будет крайне низкой. Но пойдем дальше и предположим, что все процессоры одинаковы. Проблемы кончились? Опять нет! Процессоры выполнили свою работу, но результат-то надо передать другому для продолжения процесса суммирования... а на передачу уходит время... и в это время процессоры опять простаивают...
Словом, заставить параллельную вычислительную систему или супер-ЭВМ работать с максимальной эффективность на конкретной программе это, прямо скажем, задача не из простых, поскольку **необходимо тщательное согласование структуры программ и алгоритмов с особенностями архитектуры параллельных вычислительных систем**.

Суперлинейное ускорение. При увеличении числа процессоров, например, в 2 раза задача начинает выполняться больше, чем в 2 раза. Такое возможно, если она начинается помещаться в кэш процессоров, при условии, что раньше не помещалась и приходилось обращаться к медленной памяти.

БИЛЕТ 7. Иерархия памяти, локальность вычислений, локальность использования памяти.

Регистры - самая быстрая память расположенная на роцессоре
Кэш
ОП
Диски (со своим кэшем)
Ленты

БИЛЕТ 8. Этапы решения задач на параллельных вычислительных системах.



Итак, вы приступаете к созданию параллельной программы. Желание есть, задача ясна, метод выбран, целевой компьютер, скорее всего, тоже определен. Осталось только все мысли выразить в той или иной форме, понятной для этого компьютера. Чем руководствоваться, если собственного опыта пока мало, а априорной информации о доступных технологиях параллельного программирования явно недостаточно? Наводящих соображений может быть много, но в результате вы все равно будете вынуждены пойти на компромисс, делая выбор между временем разработки программы, ее эффективностью и переносимостью, интенсивностью последующего использования программы,

необходимостью ее дальнейшего развития. Не вдаваясь в детали выбора, попробуйте для начала оценить, насколько важны для вас следующие три характеристики.

Основное назначение параллельных компьютеров — это быстро решать задачи. Если технология программирования по разным причинам не позволяет в полной мере использовать весь потенциал вычислительной системы, то нужно ли тратить усилия на ее освоение? Не будем сейчас обсуждать причины. Ясно то, что возможность создания эффективных программ является серьезным аргументом в выборе средств программирования.

Технология может давать пользователю полный контроль над использованием ресурсов вычислительной системы и ходом выполнения его программы. Для этого ему предлагается набор из нескольких сотен конструкций и функций, предназначенных "на все случаи жизни". Он может создать действительно эффективную программу, если правильно воспользуется предложенными средствами. Но захочет ли он это делать? Не стоит забывать, что он должен решать свою задачу из своей предметной области, где и своих проблем хватает. Маловероятно, что физик, химик, геолог или эколог с большой радостью захочет осваивать новую специальность, связанную с параллельным программированием. Возможность быстрого создания параллельных программ должна приниматься в расчет наравне с другими факторами.

Вычислительная техника меняется очень быстро. Предположим, что была найдена технология, позволяющая быстро создавать эффективные параллельные программы. Что произойдет через пару лет, когда появится новое поколение компьютеров? Возможных вариантов развития событий два. Первый вариант — разработанные прежде программы были "одноразовыми" и сейчас ими уже никто не интересуется. Бывает и так. Однако, как правило, в параллельные программы вкладывается слишком много средств (времени, усилий, финансовых затрат), чтобы просто так об этом забыть и начать разработку заново. Хочется перенести накопленный багаж на новую компьютерную платформу. Скорее всего, на новом компьютере старая программа рано или поздно работать будет, и даже будет давать правильный результат. Но дает ли выбранная технология гарантии сохранения эффективности параллельной программы при ее переносе с одного компьютера на другой? Скорее всего, нет. Программу для новой платформы нужно оптимизировать заново. А тут еще разработчики новой платформы предлагают вам очередную новую технологию программирования, которая опять позволит создать выдающуюся программу для данного компьютера. Программы переписываются, и так по кругу в течение многих лет.

Выбор технологии параллельного программирования — это и в самом деле вопрос не простой. Если попытаться сделать обзор средств, которые могут помочь в решении задач на параллельном компьютере, то даже поверхностный анализ приведет к списку из более 100 наименований.

В некоторых случаях выбор определяется просто. Например, вполне жизненной является ситуация, когда воспользоваться можно только тем, что установлено на доступном вам компьютере. Другой аргумент звучит так: "... все используют MPI, поэтому и я тоже буду...". Если есть возможность и желание сделать осознанный выбор, это обязательно нужно делать. Посоветуйтесь со специалистами. Проблемы в дальнейшем возникнут в любом случае, вопрос только насколько быстро и в каком объеме. Если выбор будет правильным, проблем будет меньше. Если неправильным, то тоже не отчаивайтесь, будет возможность подумать о выборе еще раз. Сделав выбор несколько раз, вы станете специалистом в данной области, забудете о своих прежних интересах, скажем, о квантовой химии или вычислительной гидродинамике. Не исключено, что в итоге вы сможете предложить свою технологию и найти ответ на центральный вопрос параллельных вычислений: "Как создавать эффективные программы для параллельных компьютеров?"

В данной главе мы рассмотрим различные подходы к программированию параллельных компьютеров. Одни широко используются на практике, другие интересны своей идеей, третьи лаконичны и выразительны. Хотелось показать широкий спектр существующих средств, но и не сводить описание каждой технологии до одного абзаца текста. Противоречивая задача. Однако надеемся, что после изучения каждого раздела вы сможете не только проводить качественное сравнение технологий, но и самостоятельно писать содержательные программы.
Знакомясь с различными системами параллельного программирования, обязательно обратите внимание на следующее обстоятельство. Если вы решаете учебные задачи или производственные задачи небольшого размера, вам почти наверняка не придется задумываться об эффективности использования параллельной вычислительной техники. В этом случае выбор системы программирования практически не имеет значения. Используйте то, что вам больше нравится. Но как только вы начнете решать большие задачи и, особенно, предельно большие многовариантные задачи, вопрос эффективности может оказаться ключевым.

Очень скоро станет ясно, что при использовании любой системы параллельного программирования желание повысить производительность вычислительной техники на вашей задаче сопровождается тем, что от вас требуется все больше и больше каких-то новых сведений о структуре задачи, программы или алгоритма. Ни одна система параллельного программирования не гарантирует высокую эффективность вычислительных процессов без предоставления дополнительных сведений.

БИЛЕТ 9. Компьютеры с общей и распределенной памятью. Две задачи параллельных вычислений.

1. Векторно-конвейерные компьютеры. Конвейерные функциональные устройства и набор векторных команд - это две особенности таких машин. В отличие от традиционного подхода, векторные команды оперируют целыми массивами независимых данных, что позволяет эффективно загружать доступные конвейеры, т.е. команда вида A=B+C может означать сложение двух массивов, а не двух чисел. Характерным представителем данного направления является семейство векторно-конвейерных компьютеров CRAY куда входят, например, CRAY EL, CRAY J90, CRAY T90 (в марте 2000 года американская компания TERA перекупила подразделение CRAY у компании Silicon Graphics, Inc.).
2. Массивно-параллельные компьютеры с распределенной памятью. Идея построения компьютеров этого класса тривиальна: возьмем серийные микропроцессоры, снабдим каждый своей локальной памятью, соединим посредством некоторой коммуникационной среды - вот и все. Достоинств у такой архитектуры масса: если нужна высокая производительность, то можно добавить еще процессоров, если ограничены финансы или заранее известна требуемая вычислительная мощность, то легко подобрать оптимальную конфигурацию и т.п.
Однако есть и решающий "минус", сводящий многие "плюсы" на нет. Дело в том, что межпроцессорное взаимодействие в компьютерах этого класса идет намного медленнее, чем происходит локальная обработка данных самими процессорами. Именно поэтому написать эффективную программу для таких компьютеров очень сложно, а для некоторых алгоритмов иногда просто невозможно. К данному классу можно отнести компьютеры Intel Paragon, IBM SP1, Parsytec, в какой-то степени IBM SP2 и CRAY T3D/T3E, хотя в этих компьютерах влияние указанного минуса значительно ослаблено. К этому же классу можно отнести и сети компьютеров, которые все чаще рассматривают как дешевую альтернативу крайне дорогим суперкомпьютерам.
3. Параллельные компьютеры с общей памятью. Вся оперативная память таких компьютеров разделяется несколькими одинаковыми процессорами. Это снимает проблемы предыдущего класса, но добавляет новые - число процессоров, имеющих доступ к общей памяти, по чисто техническим причинам нельзя сделать большим. В

данное направление входят многие современные многопроцессорные SMP-компьютеры или, например, отдельные узлы компьютеров HP Exemplar и Sun StarFire.
4. Последнее направление, строго говоря, не является самостоятельным, а скорее представляет собой комбинации предыдущих трех. Из нескольких процессоров (традиционных или векторно-конвейерных) и общей для них памяти сформируем вычислительный узел. Если полученной вычислительной мощности не достаточно, то объединим несколько узлов высокоскоростными каналами. Подобную архитектуру называют кластерной, и по такому принципу построены CRAY SV1, HP Exemplar, Sun StarFire, NEC SX-5, последние модели IBM SP2 и другие. Именно это направление является в настоящее время наиболее перспективным для конструирования компьютеров с рекордными показателями производительности.

БИЛЕТ 10. NUMA и ccNUMA архитектуры. Компьютеры Cm*, BBN Butterfly.

Оба класса компьютеров (с общей и распределенной памятью) имеют свои достоинства, которые, правда, тут же плавно перетекают в их недостатки. Для компьютеров с общей памятью проще создавать параллельные программы, но их максимальная производительность сильно ограничивается небольшим числом процессоров. А для компьютеров с распределенной памятью все наоборот. Можно ли объединить достоинства этих двух классов? Одним из возможных направлений является *проектирование компьютеров с архитектурой NUMA (Non Uniform Memory Access).*

Почему проще писать параллельные программы для компьютеров с общей памятью? Потому что есть единое адресное пространство и пользователю не нужно заниматься организацией пересылок сообщений между процессами для обмена данными. Если создать механизм, который всю совокупную физическую память компьютера позволял бы программам пользователей рассматривать как единую адресуемую память, все стало бы намного проще.

По такому пути и пошли разработчики системы Cm*, создавшие еще в конце 70-х годов прошлого века первый NUMA-компьютер. Данный компьютер состоит из набора кластеров, соединенных друг с другом через межкластерную шину. Каждый кластер объединяет процессор, контроллер памяти, модуль памяти и, быть может, некоторые устройства ввода/вывода, соединенные между собой посредством локальной шины. Когда процессору нужно выполнить операции чтения или записи, он посылает запрос с нужным адресом своему контроллеру памяти. Контроллер анализирует старшие разряды адреса, по которым и определяет, в каком модуле хранятся нужные данные. Если адрес локальный, то запрос выставляется на локальную шину, в противном случае запрос для удаленного кластера отправляется через межкластерную шину. В таком режиме программа, хранящаяся в одном модуле памяти, может выполняться любым процессором системы. Единственное различие заключается в скорости выполнения. Все локальные ссылки отрабатываются намного быстрее, чем удаленные. Поэтому и процессор того кластера, где хранится программа, выполнит ее на порядок быстрее, чем любой другой.

От этой особенности и происходит название данного класса компьютеров — компьютеры с неоднородным доступом к памяти. В этом смысле иногда говорят, что классические SMP-компьютеры обладают *архитектурой UMA (Uniform Memory Access),* обеспечивая одинаковый доступ любого процессора к любому модулю памяти.

Другим примером NUMA-компьютера являлся компьютер BBN Butterfly, который в максимальной конфигурации объединял 256 процессоров (рис. 2.16). Каждый вычислительный узел компьютера содержит процессор, локальную память и контроллер памяти, который определяет, является ли запрос к памяти локальным или его необходимо передать удаленному узлу через коммутатор Butterfly. С точки зрения программиста память является единой общей памятью, удаленные ссылки в которой реализуются немного дольше локальных (приблизительно 6 мкс для удаленных против 2 мкс для локальных).

По пути построения больших NUMA-компьютеров можно было бы смело идти вперед, если бы не одна неожиданная проблема — кэш-память отдельных процессоров. Кэш-память, которая помогает значительно ускорить работу отдельных процессоров, для многопроцессорных систем оказывается узким местом. В процессорах первых NUMA-компьютеров кэш-памяти не было, поэтому не было и такой проблемы. Но для современных микропроцессоров кэш является неотъемлемой составной частью. Причину нашего беспокойства очень легко объяснить. Предположим, что процессор P1 сохранил значение *x* в ячейке *q,* а затем процессор P*2* хочет прочитать содержимое той же ячейки *q*. Что получит процессор *P2?* Конечно же, всем бы хотелось, чтобы он получил значение *x,* но как он его получит, если *x* попало в кэш процессора *P1?* Эта проблема носит название проблемы согласования содержимого кэш-памяти *(cache coherence problem, проблема когерентности кэшей).* Указанная проблема актуальна и для современных SMP-компьютеров, кэш процессоров которых также может вызвать несогласованность в использовании данных.

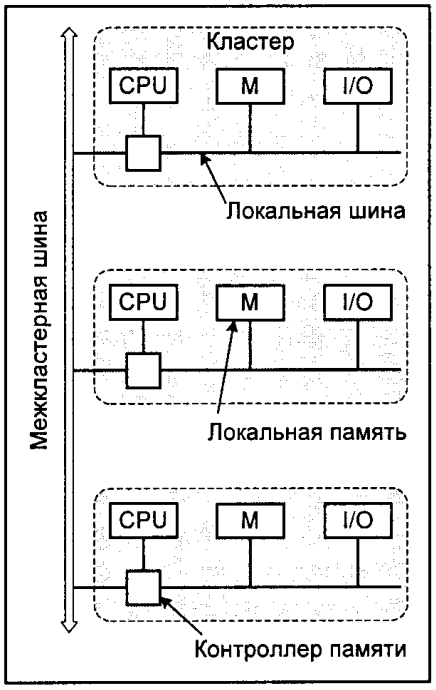


Рис. 2.15. Схема вычислительной системы Cm*

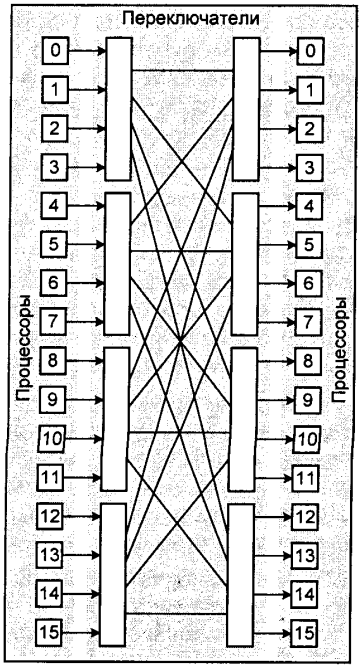


Рис. 2.16 Схема вычислительной системы BBN Butterfly

Для решения данной проблемы разработана специальная модификация NUMA-архитектуры — ccNUMA (cache coherent NUMA). Не будем сейчас вдаваться в технические подробности множества протоколов, которые обеспечивают согласованность содержимого всех кэшей. Важно, что эта проблема решается и не ложится на плечи пользователей. Для пользователей важнее другой вопрос, насколько "неоднородна" архитектура NUMA? Если обращение к памяти другого узла требует на 5—10% больше времени, чем обращение к своей памяти, то это может и не вызвать никаких вопросов. Большинство пользователей будут относиться к такой системе, как к UMA (SMP), и практически все разработанные для SMP программы будут работать достаточно хорошо. Однако для современных NUMA систем это не так, и разница времени локального и удаленного доступа лежит в промежутке 200—700%. При такой разнице в скорости доступа для обеспечения должной эффективности выполнения программ следует позаботиться о правильном расположении требуемых данных.

На основе архитектуры ccNUMA в настоящее время выпускается множество реальных систем, расширяющих возможности традиционных компьютеров с общей памятью. При этом, если конфигурации SMP серверов от ведущих производителей содержат 16—32—64

процессора, то их расширения с архитектурой ccNUMA уже объединяют до 256 процессоров и больше.

БИЛЕТ 11. Общая структура компьютера Hewlett-Packard Superdome.

Проведем исследование архитектуры параллельных компьютеров с общей памятью на примере вычислительной системы Hewlett-Packard Superdome. Компьютер появился в 2000 году, а в ноябрьской редакции списка Top500 2001 года им уже были заняты 147 позиций.

Компьютер HP Superdome в стандартной комплектации может объединять от 2 до 64 процессоров с возможностью последующего расширения системы. Все процессоры имеют доступ к общей памяти, организованной в соответствии с архитектурой ccNUMA. Это означает, что, во-первых, все процессы могут работать в едином адресном пространстве, адресуя любой байт памяти посредством обычных операций чтения/записи. Во-вторых, доступ к локальной памяти в системе будет идти немного быстрее, чем доступ к удаленной памяти. В-третьих, проблемы возможного несоответствия данных, вызванные кэш-памятью процессоров, решены на уровне аппаратуры.

В максимальной конфигурации Superdome может содержать до 256 Гбайт оперативной памяти. Ближайшие планы компании — реализовать возможность наращивания памяти компьютера до 1 Тбайта.

Архитектура компьютера спроектирована таким образом, что в ней могут использоваться несколько типов микропроцессоров. Это, конечно же, традиционные для вычислительных систем Hewlett-Packard процессоры семейства PA: PA-8600 и PA-8700. Вместе с тем, система полностью подготовлена и к использованию процессоров следующего поколения с архитектурой IA-64, разработанной совместно компаниями HP и Intel. При замене существующих процессоров на процессоры IA-64 гарантируется двоичная совместимость приложений на системном уровне. В дальнейшем, если другого не оговорено, будем рассматривать конфигурации HP Superdome на базе процессора PA-8700.

Основу архитектуры компьютера HP Superdome составляют вычислительные ячейки (cells), связанные иерархической системой переключателей. Каждая ячейка является симметричным мультипроцессором, реализованным на одной плате, в котором есть все необходимые компоненты (рис. 3.12):

* процессоры (до 4-х);
* оперативная память (до 16 Гбайт);
* контроллер ячейки;
* преобразователи питания;
* связь с подсистемой ввода/вывода (опционально).

Интересно, что ячейки Superdome во многом похожи на аналогичные архитектурные элементы других современных ccNUMA компьютеров. В Superdome таким элементом является ячейка, в семействе SGI Origin 3x00 это узел (node), а в компьютерах серии Compaq AlphaServer GS320 — QBB (Quad Building Block). Во всех системах в каждом элементе содержится по четыре процессора.

Центральное место в архитектуре ячейки Superdome занимает контроллер ячейки. Несмотря на столь обыденное название, контроллер — это сложнейшее устройство, состоящее из 24 миллионов транзисторов. Для каждого процессора ячейки есть собственный порт в контроллере. Обмен данными между каждым процессором и контроллером идет со скоростью 2 Гбайт/с.

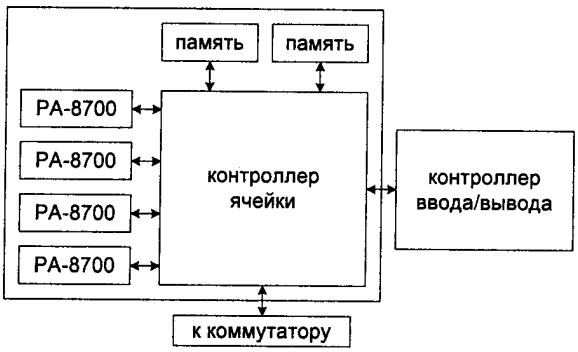


Рис. 3.12. Структура ячейки компьютера HP Superdome

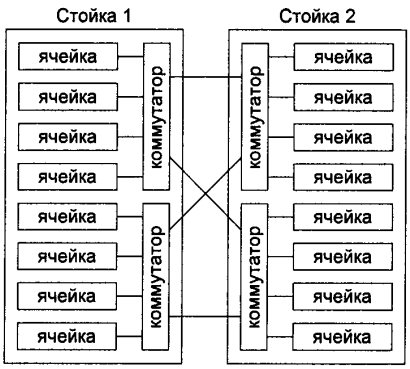


Рис. 3.13. Общая структура компьютера HP Superdome

Память ячейки имеет емкость от 2 до 16 Гбайт. Конструктивно она разделена на два банка, каждый из которых имеет свой порт в контроллере ячейки. Скорость обмена данными между контроллером и каждым банком составляет 2 Гбайт/с, что дает суммарную пропускную способность тракта контроллер—память 4 Гбайт/с. Соединение контроллера ячейки с контроллером устройств ввода/вывода (12 слотов PCI) устанавливается опционально.

Один порт контроллера ячейки всегда связан с внешним коммутатором. Он предназначен для обмена процессоров ячейки с другими процессорами системы. Скорость работы этого порта равна 8 Гбайт/с.

Выполняя интерфейсные функции между процессорами, памятью, другими ячейками и внешним миром, контроллер ячейки отвечает и за когерентность кэш-памяти процессоров.

Ячейка – это базовый четырехпроцессорный блок компьютера. В 64-процессорной конфигурации Superdome состоит из двух стоек, в каждой стойке по 32 процессора (рис. 3.13).

Каждая стойка содержит по два восьмипортовых неблокирующих коммутатора. Все порты коммутаторов работают со скоростью 8 Гбайт/с. К каждому коммутатору подключаются четыре ячейки. Три порта коммутатора задействованы для связи с другими коммутаторами системы (один в этой же стойке, и два коммутатора— в другой). Оставшийся порт зарезервирован для связи с другими системами HP Superdome, что дает потенциальную возможность для формирования многоузловой конфигурации компьютера с общим числом процессоров больше 64.

Одним из центральных вопросов любой вычислительной системы с архитектурой ccNUMA является разница во времени при обращении процессора к локальным и удаленным ячейкам памяти. В идеале хотелось бы, чтобы этой разницы, как в SMP-компьютере, не было вовсе. Однако в таком случае система заведомо будет плохо масштабируемой. В компьютере HP Superdome возможны три вида задержек при обращении процессора к памяти, являющихся своего рода платой за высокую масштабируемость системы в целом:

процессор и память располагаются в одной ячейке; в этом случае задержка минимальна;

процессор и память располагаются в разных ячейках, но обе эти ячейки подсоединены к одному и тому же коммутатору;

процессор и память располагаются в разных ячейках, причем обе эти ячейки подсоединены к разным коммутаторам; в этом случае запрос должен пройти через два коммутатора и задержки будут максимальными.

Компьютер HP Superdome имеет массу интересных особенностей. В частности, программно-аппаратная среда компьютера позволяет его настроить различным образом. Superdome может быть классическим единым компьютером с общей памятью. Однако его можно сконфигурировать и таким образом, что он будет являться совокупностью независимых разделов (nPartitions), работающих под различными операционными системами, в частности, под HP UX, Linux и Windows 2000. Организация эффективной

работы с большим числом внешних устройств, возможности "горячей" замены всех основных компонентов аппаратуры, резервирование, мониторинг базовых параметров — все это останется за рамками нашего обсуждения.

БИЛЕТ 12. Причины уменьшения быстродействия компьютера Hewlett-Packard Superdome

В заключение, как и в предыдущем параграфе, выделим те особенности вычислительных систем с общей памятью, которые снижают их производительность на реальных программах. **Закон Амдала** носит универсальный характер, поэтому он упоминается вместе со всеми параллельными системами. Не являются исключением и компьютеры с общей памятью. Если в программе 20% всех операций должны выполняться строго последовательно, то ускорения больше 5 получить нельзя вне зависимости от числа использованных процессоров (влияние кэшпамяти сейчас не рассматривается). Это нужно учитывать и перед адаптацией старой последовательной программы к такой архитектуре, и в процессе проектирования нового параллельного кода.

Для компьютеров с общей памятью дополнительно следует принять в расчет и такие соображения. Наличие физической общей памяти стимулирует к использованию моделей параллельных программ также с общей памятью. Это вполне естественно и оправданно. Однако в этом случае возникают дополнительные участки последовательного кода, связанные с синхронизацией доступа к общим данным, например, критические секции. Относительно подобных конструкций в описании соответствующей технологии программирования может и не быть никакого предостережения, однако реально эти фрагменты будут последовательными участками кода.

Работа с памятью является очень тонким местом в системах данного класса. Одну из причин снижения производительности — **неоднородность доступа к памяти**, мы уже обсуждали. Степень неоднородности на уровне 5—10% серьезных проблем не создаст. Однако разница во времени доступа к локальной и удаленной памяти в несколько раз потребует от пользователя очень аккуратного программирования. В этом случае ему придется решать вопросы, аналогичные распределению данных для систем с распределенной памятью. Другую причину — **конфликты при обращении к памяти** — мы детально не разбирали, но она также характерна для многих SMP-систем.

Наличие кэш-памяти у каждого процессора тоже привносит свои дополнительные особенности. Наиболее существенная из них состоит в **необходимости обеспечения согласованности содержимого кэш-памяти**. Отсюда появились и первые две буквы в аббревиатуре ccNUMA. Чем реже вовлекается аппаратура в решение этой проблемы, тем меньше накладных расходов сопровождает выполнение программы. По этой же причине во многих системах с общей памятью существует режим выполнения параллельной программы с привязкой процессов к процессорам.

Сбалансированность вычислительной нагрузки также характерна для параллельных систем, как и закон Амдала. В случае систем с общей памятью ситуация упрощается тем, что практически всегда системы являются однородными. Они содержат одинаковые процессоры, поэтому о сложной стратегии распределения работы речь, как правило, не идет.

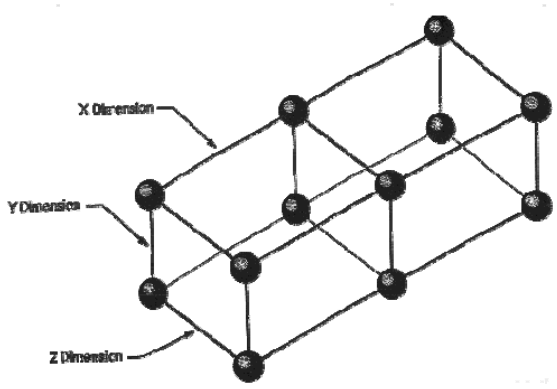
Любой современный процессор имеет сложную архитектуру, объединяющую и несколько уровней памяти, и множество функциональных устройств. **Реальная производительность отдельного процессора** может отличаться от его же пиковой в десятки раз. Чем выше степень использования возможностей каждого процессора, тем выше общая производительность вычислительной системы.

БИЛЕТ 13. Общая структура компьютера CRAY T3E: вычислительные узлы и процессорные элементы.

Компьютер CRAY T3D - это массивно-параллельный компьютер с распределенной памятью, объединяющий от 32 до 2048 процессоров. Распределенность памяти означает

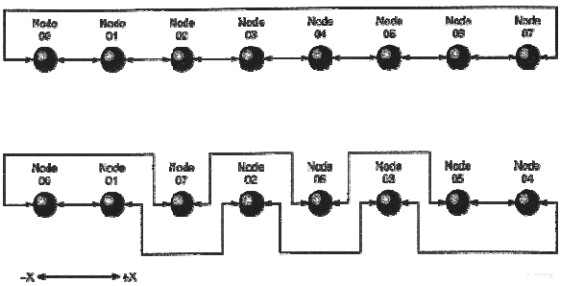
то, что каждый процессор имеет непосредственный доступ только к своей локальной памяти, а доступ к данным, расположенным в памяти других процессоров, выполняется другими, более сложными способами.
CRAY T3D подключается к хост-компьютеру (главному или ведущему), роль которого, в частности, может исполнять CRAY Y-MP C90. Вся предварительная обработка и подготовка программ, выполняемых на CRAY T3D, проходит на хосте (например, компиляция). Связь хост-машины и T3D идет через высокоскоростной канал передачи данных с производительностью 200 Мбайт/с.
Массивно-параллельный компьютер CRAY T3D работает на тактовой частоте 150MHz и имеет в своем составе три основные компоненты: сеть межпроцессорного взаимодействия (или по-другому коммуникационную сеть), вычислительные узлы и узлы ввода/вывода.
Вычислительный узел состоит из двух процессорных элементов (ПЭ), сетевого интерфейса, контроллера блочных передач. Оба процессорных элемента, входящие в состав вычислительного узла, идентичны и могут работать независимо друг от друга.
Процессорный элемент. Каждый ПЭ содержит микропроцессор, локальную память и некоторые вспомогательные схемы.
Микропроцессор - это 64-х разрядный RISC (Reduced Instruction Set Computer) процессор ALPHA фирмы DEC, работающий на тактовой частоте 150 MHz. Микропроцессор имеет внутреннюю кэш-память команд и кэш-память данных.
Объем локальной памяти ПЭ - 8 Мслов. Локальная память каждого процессорного элемента является частью физически распределенной, но логически разделяемой (или общей), памяти всего компьютера. В самом деле, память физически распределена, так как каждый ПЭ содержит свою локальную память. В тоже время, память разделяется всеми ПЭ, так как каждый ПЭ может обращаться к памяти любого другого ПЭ, не прерывая его работы.
Обращение к памяти другого ПЭ лишь в 6 раз медленнее, чем обращение к своей собственной локальной памяти.
Сетевой интерфейс формирует передачи перед посылкой через коммуникационную сеть другим вычислительным узлам или узлам ввода/вывода, а также принимает приходящие сообщения и распределяет их между двумя процессорными элементами узла.
Контроллер блочных передач - это контроллер асинхронного прямого доступа в память, который помогает перераспределять данные, расположенные в локальной памяти разных ПЭ компьютера CRAY T3D, без прерывания работы самих ПЭ.

БИЛЕТ 14. Общая структура компьютера CRAY T3E: коммуникационная сеть.



Коммуникационная сеть

Коммуникационная сеть обеспечивает передачу информации между вычислительными узлами и узлами ввода/вывода с максимальной скоростью в 140М байт/с. Сеть образует трехмерную решетку, соединяя сетевые маршрутизаторы узлов в направлениях *X*, *Y*, *Z*. Каждая элементарная связь между двумя узлами - это два однонаправленных канала передачи данных, что допускает одновременный обмен данными в противоположных направлениях.



Топология сети, чередование вычислительных узлов

Коммуникационная сеть компьютера CRAY T3D организована в виде двунаправленного трехмерного тора, что имеет свои преимущества перед другими способами организации связи:

быстрая связь граничных узлов и небольшое среднее число перемещений по тору при взаимодействии разных ПЭ: максимальное расстояние в сети для конфигурации из 128 ПЭ равно 6, а для 2048 ПЭ равно 12;

возможность выбора другого маршрута для обхода поврежденных связей.

Все узлы в коммуникационной сети в размерностях *X* и *Z* расположены с чередованием, что позволяет минимизировать длину максимального физического соединения между ПЭ.

Маршрутизация в сети и сетевые маршрутизаторы.

При выборе маршрута для обмена данными между двумя узлами сетевые маршрутизаторы всегда сначала выполняют смещение по размерности *X*, затем по *Y*, а в конце по *Z*. Так как смещение может быть как положительным, так и отрицательным, то этот механизм помогает минимизировать число перемещений по сети и обойти поврежденные связи. Сетевые маршрутизаторы каждого вычислительного узла определяют путь перемещения каждого пакета и могут осуществлять параллельный транзит данных по каждому из трех измерений *X*, *Y*, *Z*.

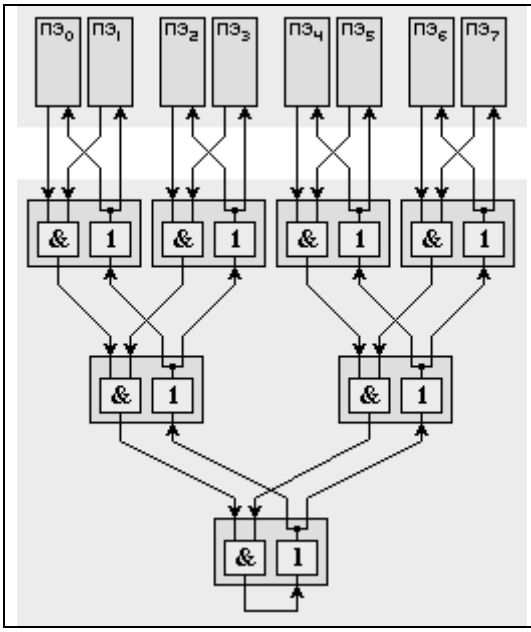
Нумерация вычислительных узлов.

Каждому ПЭ в системе присвоен уникальный *физический номер*, определяющий его физическое расположение, который и используется непосредственно аппаратурой. Не обязательно все физические ПЭ принимают участие в формировании логической конфигурации компьютера. Например, 512-процессорная конфигурация компьютера CRAY T3D реально содержит 520 физических ПЭ, 8 из которых находятся в резерве. Каждому физическому ПЭ присваивается *логический номер*, определяющий его расположение в логической конфигурации компьютера, которая уже и образует трехмерный тор.

Каждой программе пользователя из трехмерной решетки вычислительных узлов выделяется отдельный раздел, имеющий форму прямоугольного параллелепипеда, на котором работает только данная программа (не считая компонент ОС). Для последовательной нумерации ПЭ, выделенных пользователю, вводится *виртуальная нумерация*.

БИЛЕТ 15. Общая структура компьютера CRAY T3E: аппаратная поддержка синхронизации параллельных процессов.

Для поддержки синхронизации процессорных элементов предусмотрена аппаратная реализация одного из наиболее «тяжелых» видов синхронизации – барьеров синхронизации. Барьер – это точка в программе, при достижении которой каждый процессор должен ждать до тех пор, пока остальные также не дойдут до барьера, и лишь после этого момента все процессы могут продолжать работу дальше.



В схемах поддержки каждого ПЭ предусмотрены два 8-ми разрядных регистра, причем каждый разряд регистров соединен со своей независимой цепью реализации барьера (всего 16 независимых цепей). Каждая цепь строится на основе схем AND и ДУБЛИРОВАНИЕ (1-2). До барьера соответствующие разряды на всех ПЭ обнуляются, а как только процесс на ПЭ доходит до барьера, то записывает в свой разряд единицу. На выходе схемы AND появляется единица только в том случае, когда на обоих входах выставлены 1. Устройство 1-2 просто дублирует свой вход на выходы:

Если схемы AND заменить на OR, то получится цепь для реализации механизма «эврика»: как только один ПЭ выставил значение 1, эта единица распространяется всем ПЭ, сигнализируя о некотором событии на ПЭ. Это исключительно полезно, например, в задачах поиска.

БИЛЕТ 16. Вычислительные кластеры: узлы, коммуникационная сеть (латентность, пропускная способность), способы построения.

Различных вариантов построения кластеров очень много. Одно из существенных различий лежит в используемой сетевой технологии, выбор которой определяется, прежде всего, классом решаемых задач.

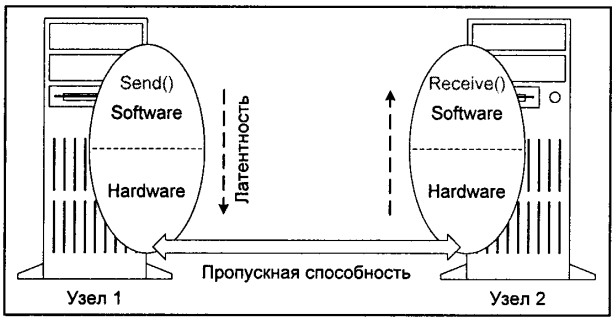


Рис. 3.19. Латентность и пропускная способность коммуникационной среды

Какими же числовыми характеристиками выражается производительность коммуникационных сетей в кластерных системах? Необходимых пользователю характеристик две: латентность и пропускная способность сети. *Латентность* — это время начальной задержки при посылке сообщений. *Пропускная способность сети* определяется скоростью передачи информации по каналам связи (рис. 3.19). Если в программе много маленьких сообщений, то сильно скажется латентность. Если сообщения передаются большими порциями, то важна высокая пропускная способность каналов связи. Из-за латентности максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной.

На практике пользователям не столько важны заявляемые производителем пиковые характеристики, сколько реальные показатели, достигаемые на уровне приложений. После вызова пользователем функции посылки сообщения Send() сообщение последовательно проходит через целый набор слоев, определяемых особенностями организации

программного обеспечения и аппаратуры. Этим, в частности, определяются и множество вариаций на тему латентности реальных систем. Установили MPI на компьютере плохо, латентность будет большая, купили дешевую сетевую карту от неизвестного производителя, ждите дальнейших сюрпризов.

В заключение параграфа давайте попробуем и для данного класса компьютеров выделить факторы, снижающие производительность вычислительных систем с распределенной памятью на реальных программах.

Начнем с уже упоминавшегося ранее *закона Амдала.* Для компьютеров данного класса он играет очень большую роль. В самом деле, если предположить, что в программе есть лишь 2% последовательных операций, то рассчитывать на более чем 50-кратное ускорение работы программы не приходится. Теперь попробуйте критически взглянуть на свою программу. Скорее всего, в ней есть инициализация, операции ввода/вывода, какие-то сугубо последовательные участки. Оцените их долю на фоне всей программы и на мгновенье предположите, что вы получили доступ к вычислительной системе из 1000 процессоров. После вычисления верхней границы для ускорения программы на такой системе, думаем, станет ясно, что недооценивать влияние закона Амдала никак нельзя.

Поскольку компьютеры данного класса имеют распределенную память, то взаимодействие процессоров между собой осуществляется с помощью передачи сообщений. Отсюда два других замедляющих фактора — *латентность* и *скорость передачи данных* по каналам коммуникационной среды. В зависимости от коммуникационной структуры программы степень влияния этих факторов может сильно меняться.

Если аппаратура или программное обеспечение не поддерживают возможности *асинхронной посылки сообщений* на фоне вычислений, то возникнут неизбежные накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов.

Для достижения эффективной параллельной обработки необходимо добиться *максимально равномерной загрузки всех процессоров.* Если равномерности нет, то часть процессоров неизбежно будет простаивать, ожидая остальных, хотя в это время они вполне могли бы выполнять полезную работу. Данная проблема решается проще, если вычислительная система однородна. Очень большие трудности возникают при переходе на неоднородные системы, в которых есть значительное различие либо между вычислительными узлами, либо между каналами связи.

Существенный фактор — это *реальная производительность одного процессора* вычислительной системы. Разные модели микропроцессоров могут поддерживать несколько уровней кэш-памяти, иметь специализированные функциональные устройства и т. п. Возьмем хотя бы иерархию памяти компьютера Cray T3E: регистры процессора, кэш-память 1-го уровня, кэш-память 2-го *уровня,* локальная память процессора, *удаленная память другого* процессора. Эффективное использование такой структуры требует особого внимания *при выборе подхода к решению задачи.*

Дополнительно каждый микропроцессор может иметь элементы векторно-конвейерной архитектуры. В этом случае ему будут присущи многие факторы, которые мы обсуждали в конце *§3.2.*

К сожалению, как и прежде, на работе каждой конкретной программы в той или иной мере сказываются все эти факторы. Однако в отличие от компьютеров других классов, суммарное воздействие изложенных здесь факторов может снизить реальную производительность не в десятки, а в сотни и даже тысячи раз по сравнению с пиковой. Потенциал компьютеров этого класса огромен, добиться на них можно очень многого.

Крайняя точка — Интернет. Его тоже можно рассматривать как компьютер с распределенной памятью. Причем, как самый мощный в мире компьютер.

БИЛЕТ 17. Причины уменьшения производительности компьютеров с распределенной памятью.

Начнем с уже упоминавшегося закона Амдала. Для массивно-параллельных компьютеров он играет еще большую роль, чем для векторно-конвейерных. В самом деле, в таблице 1 показано, на какое максимальное ускорение работы программы можно рассчитывать в зависимости от доли последовательных вычислений и числа доступных процессоров. Предполагается, что параллельная секция может быть выполнена без каких-либо дополнительных накладных расходов. Так как в программе всегда присутствует инициализация, ввод/вывод и некоторые сугубо последовательные действия, то недооценивать данный фактор никак нельзя - практически вся программа должна исполняться в параллельном режиме, что можно обеспечить только после анализа всей (!) программы.

Число ПЭ	Доля последовательных вычислений				
	50%	25%	10%	5%	2%
2	1.33	1.60	1.82	1.90	1.96
8	1.78	2.91	4.71	5.93	7.02
32	1.94	3.66	7.80	12.55	19.75
512	1.99	3.97	9.83	19.28	45.63
2048	2.00	3.99	9.96	19.82	48.83

Табл. 1. Максимальное ускорение работы программы в зависимости от доли последовательных вычислений и числа используемых процессоров.

Поскольку CRAY T3D - это компьютер с распределенной памятью, то взаимодействие процессоров, в основном, осуществляется посредством передачи сообщений друг другу. Отсюда два других замедляющих фактора - **время инициализации посылки сообщения** (латентность) и собственно **время передачи сообщения по сети**. Максимальная скорость передачи достигается на больших сообщениях, когда латентность, возникающая лишь в начале, не столь заметна на фоне непосредственно передачи данных.

Возможность асинхронной посылки сообщений и вычислений. Если или аппаратура, или программное обеспечение не поддерживают возможности проводить вычислений на фоне пересылок, то возникнут неизбежные накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов.

Для достижения эффективной параллельной обработки необходимо добиться **равномерной загрузки всех процессоров**. Если равномерности нет, то часть процессоров неизбежно будет простаивать, ожидая остальных, хотя в этот момент они могли бы выполнять полезную работу. Иногда равномерность получается автоматически, например, при обработке прямоугольных матриц достаточно большого размера, однако уже при переходе к треугольным матрицам добиться хорошей равномерности не так просто.

Если один процессор должен вычислить некоторые данные, которые нужны другому процессору, и если второй процесс первым дойдет до точки приема соответствующего сообщения, то он с неизбежностью будет простаивать, ожидая передачи. Для того чтобы минимизировать время ожидание прихода сообщения первый процесс должен отправить требуемые данные как можно раньше, отложив независящую от них работу на потом, а второй процесс должен выполнить максимум работы, не требующей ожидаемой передачи, прежде, чем выходить на точку приема сообщения.

Чтобы не сложилось совсем плохого впечатления о массивно-параллельных компьютерах, надо заканчивать с негативными факторами, потому последний фактор - это **реальная производительность одного процессора**. Разные модели микропроцессоров могут поддерживать несколько уровней кэш-памяти, иметь специализированные функциональные устройства, регистровую структуру и т.п. Каждый микропроцессор, в конце концов, может иметь векторно-конвейерную архитектуру, и в этом случае ему

присущи практически все те факторы, которые мы обсуждали в лекции, посвященной особенностям программирования векторно-конвейерных компьютеров.
К сожалению, на работе каждой конкретной программы сказываются, в той или иной мере, все эти факторы одновременно, дополнительно усугубляя ситуацию с эффективностью параллельных программ. Однако в отличие от векторно-конвейерных компьютеров все изложенные здесь факторы, за исключением, быть может, последнего, могут снизить производительность не в десятки, а в сотни и даже тысячи раз по сравнению с пиковыми показателями производительности компьютера. Добиться на этих компьютерах, в принципе, можно многого, но усилий это может потребовать во многих случаях очень больших.

БИЛЕТ 18. Метакомпьютер и метакомпьютинг. Отличительные свойства распределенных вычислительных сред.

Вычислительный кластер можно собрать практически в любой лаборатории, отталкиваясь от потребностей в вычислительной мощности и доступного бюджета. Для целого класса задач, где не предполагается тесного взаимодействия между параллельными процессами, решение на основе обычных рабочих станций и сети Fast Ethernet будет вполне эффективным. Но чем такое решение отличается от обычной локальной сети современного предприятия? С точки зрения прикладного программиста почти ничем. Если у него появится возможность использования подобной сети для решения своих задач, то для него такая конфигурация и будет параллельным компьютером.

Продолжая идею дальше, любые вычислительные устройства можно считать параллельной вычислительной системой, если они работают одновременно и их можно использовать для решения одной задачи.

В этом смысле уникальные возможности дает сеть Интернет, которую можно рассматривать как самый большой компьютер в мире. Никакая вычислительная система не может сравниться ни по пиковой производительности, ни по объему оперативной или дисковой памяти с теми суммарными ресурсами, которыми обладают компьютеры, подключенные к Интернету. Компьютер, состоящий из компьютеров, своего рода метакомпьютер. Отсюда происходит и специальное название для процесса организации вычислений на такой вычислительной системе — метакомпъютинг. В принципе, совершенно не обязательно рассматривать именно Интернет в качестве коммуникационной среды метакомпьютера, эту роль может выполнять любая сетевая технология. В данном случае для нас важен принцип, а возможностей для технической реализации сейчас существует предостаточно. Вместе с тем, к Интернету всегда был и будет особый интерес, поскольку никакая отдельная вычислительная система не сравнится по своей мощности с потенциальными возможностями глобальной сети.

Конструктивные идеи использования распределенных вычислительных ресурсов для решения сложных задач появились относительно недавно. Первые прототипы реальных систем метакомпьютинга стали доступными с середины 90-х годов прошлого века. Некоторые системы претендовали на универсальность, часть из них была сразу ориентирована на решение конкретных задач, где-то ставка делалась на использование выделенных высокопроизводительных сетей и специальных протоколов, а где-то за основу брались обычные каналы и работа по протоколу HTTP..

Прогресс в сетевых технологиях последних лет колоссален. Гигабитные линии связи между компьютерами, разнесенными на сотни километров, становятся обычной реальностью. Объединив различные вычислительные системы в рамках единой сети, можно сформировать специальную вычислительную среду. Какие-то компьютеры могут подключаться или отключаться, но, с точки зрения пользователя, эта виртуальная среда является единым метакомпьютером. Работая в такой среде, пользователь лишь выдает задание на решение задачи, а остальное метакомпьютер делает сам: ищет доступные вычислительные ресурсы, отслеживает их работоспособность, осуществляет передачу

данных, если требуется, то выполняет преобразование данных в формат компьютера, на котором будет выполняться задача, и т. п. Пользователь может даже и не узнать, ресурсы какого именно компьютера были ему предоставлены. А, по большому счету, часто ли вам это нужно знать? Если потребовались вычислительные мощности для решения задачи, то вы подключаетесь к метакомпьютеру, выдаете задание и получаете результат. Все.

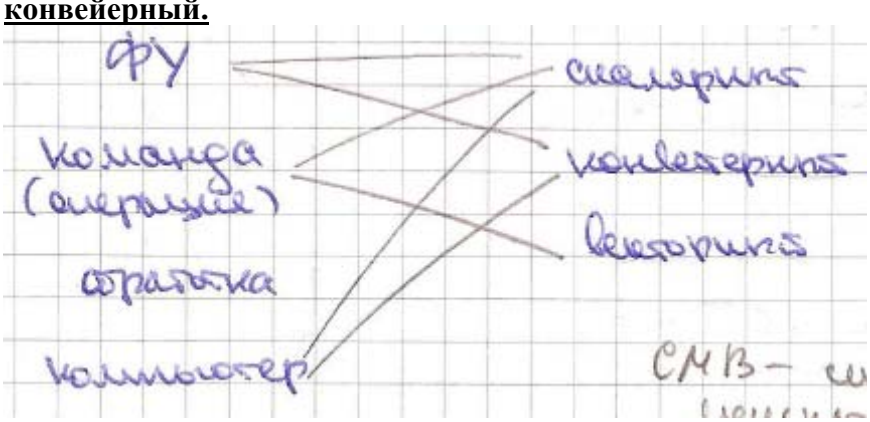
В отличие от традиционного компьютера метакомпьютер имеет целый набор присущих только ему особенностей:

* метакомпьютер обладает **огромными ресурсами**, которые несравнимы с ресурсами обычных компьютеров. Это касается практически всех параметров: число доступных процессоров, объем памяти, число активных приложений, пользователей и т. п.;
* метакомпьютер является **распределенным** по своей природе. Компоненты метакомпьютера могут быть удалены друг от друга на сотни и тысячи километров, что неизбежно вызовет большую латентность и, следовательно, скажется на оперативности их взаимодействия;
* метакомпьютер может **динамически менять конфигурацию**. Какие-то компьютеры к нему подсоединяются и делегируют права на использование своих ресурсов, какие-то отключаются и становятся недоступными. Но для пользователя работа с метакомпьютером должна быть прозрачной. Задача системы поддержки работы метакомпьютера состоит в поиске подходящих ресурсов, проверке их работоспособности, в распределении поступающих задач вне зависимости от текущей конфигурации метакомпьютера в целом;
* метакомпьютер **неоднороден**. При распределении заданий нужно учитывать особенности операционных систем, входящих в его состав. Разные системы поддерживают различные системы команд и форматы представления данных. Различные системы в разное время могут иметь различную загрузку, связь с вычислительными системами идет по каналам с различной пропускной способностью. Наконец, в состав метакомпьютера могут входить системы с принципиально различной архитектурой;
* метакомпьютер **объединяет ресурсы** различных организаций. Политика доступа и использования конкретных ресурсов может сильно меняться в зависимости от их принадлежности к той или иной организации. Метакомпьютер не принадлежит никому, поэтому политика его администрирования может быть определена лишь в самых общих чертах. Вместе с тем, согласованность работы огромного числа составных частей метакомпьютера предполагает обязательную стандартизацию работы всех его служб и сервисов.

Говоря о метакомпьютере, следует четко представлять, что речь идет не только об аппаратной части и не столько об аппаратной части, сколько о его инфраструктуре. В комплексе должны рассматриваться такие вопросы, как средства и модели программирования, распределение и диспетчеризация заданий, технологии организации доступа к метакомпьютеру, интерфейс с пользователями, безопасность, надежность, политика администрирования, средства доступа и технологии распределенного хранения данных, мониторинг состояния различных подсистем метакомпьютера и многие другие. Представьте себе, как заставить десятки миллионов различных электронных устройств, составляющих метакомпьютер, работать согласованно над заданиями десятков тысяч пользователей в течение продолжительного времени? Фантастически сложная задача, масштабное решение которой было невозможным еще десять лет назад.

В настоящее время идет активное обсуждение различных стратегий построения метакомпьютера. Однако многие вопросы до сих пор недостаточно проработаны, часть предложенных технологий еще находится на стадии апробации, не всегда используется единая терминология. Ситуация в данной области развивается чрезвычайно быстро.

19. Соотношение между понятиями: функциональное устройство, команда (операция), обработка, компьютер и их характеристиками: скалярный, векторный, конвейерный.



20. Общая структура компьютера CRAY C90, структура памяти.

С появлением в 1976 году компьютера Cray-1 началась история векторно-конвейерных вычислительных систем. Архитектура оказалась настолько удачной, что компьютер дал начало целому семейству машин, а его название стало нарицательным для обозначения сверхмощной вычислительной техники. Поскольку компьютеры этого семейства по праву считаются классическими представителями мира суперкомпьютеров, с них мы и начнем изучение различных классов архитектур.

В качестве объекта для детального изучения возьмем компьютер Cray C90, в архитектуре которого есть все характерные особенности компьютеров данного класса. Его последователь Cray T90 имеет такую же структуру, отличаясь лишь некоторыми количественными характеристиками. Описание компьютера будем вести с той степенью детальности, которая необходима для выделения ключевых особенностей и узких мест архитектуры. Знание именно этих параметров нам понадобится позднее для анализа эффективности функционирования реальных параллельных программ.

Итак, Cray C90 — это векторно-конвейерный компьютер, появившийся на рынке вычислительной техники в самом начале 90-х годов прошлого века. В максимальной конфигурации Cray C90 содержит 16 процессоров, работающих над общей памятью. Время такта компьютера равно 4,1 не, что соответствует тактовой частоте почти 250 МГц. На рис. 3.8 показана общая схема данного компьютера с более детальным представлением структуры одного процессора. Поскольку все процессоры одинаковы, то не имеет значения, какой именно процессор изображать детально.

Все процессоры компьютера Cray C90 не только одинаковы, но и равноправны по отношению ко всем разделяемым ресурсам: памяти, секции ввода/вывода и секции межпроцессорного взаимодействия. Рассмотрим кратко их особенности.

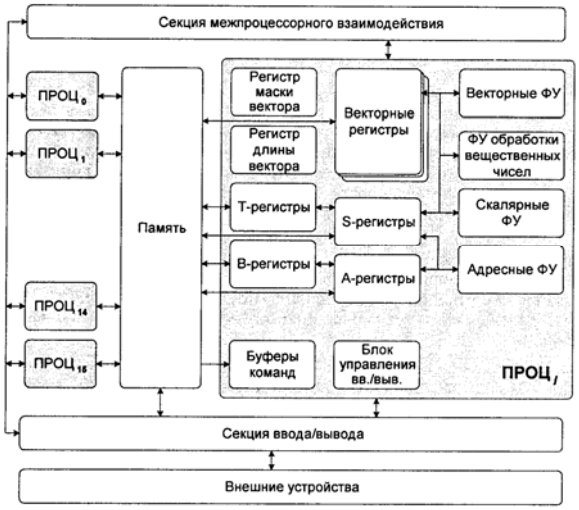


Рис. 3.8. Общая схема компьютера Cray C90

Структура оперативной памяти.

Оперативная память этого компьютера разделяется всеми процессорами и секцией ввода/вывода. Каждое слово состоит из 80-ти разрядов: 64 для хранения данных и 16 для коррекции ошибок. Для увеличения скорости выборки данных память разделена на множество банков, которые могут работать одновременно.

Каждый процессор имеет доступ к ОП через четыре порта с пропускной способностью два слова за один такт каждый, причем один из портов всегда связан с секций ввода/вывода и по крайней мере один из портов всегда выделен под операцию записи.

В максимальной конфигурации вся память разделена на 8 секций, каждая секция на 8 подсекций, каждая подсекция на 16 банков. Адреса идут с чередованием по каждому из данных параметров:

```
адрес 0 - в 0-й секции, 0-подсекции, 0-м банке,
адрес 1 - в 1-й секции, 0-подсекции, 0-м банке,
адрес 2 - в 2-й секции, 0-подсекции, 0-м банке,
  ...
адрес 8 - в 0-й секции, 1-подсекции, 0-м банке,
адрес 9 - в 1-й секции, 1-подсекции, 0-м банке,
  ...
адрес 63 - в 7-й секции, 7-подсекции, 0-м банке,
адрес 64 - в 0-й секции, 0-подсекции, 1-м банке,
адрес 65 - в 1-й секции, 0-подсекции, 1-м банке,
  ...
```

При одновременном обращении к одной и той же секции из разных портов возникает задержка в 1 такт, а при обращении к одной и той же подсекции одной секции задержка варьируется от 1 до 6 тактов. При выборке последовательно расположенных данных или при выборке с любым нечетным шагом конфликтов не возникает.

Секция ввода/вывода

Компьютер поддерживает три типа каналов, которые различаются скоростью передачи:

- Low-speed (LOSP) channels - 6 Mbytes/s
- High-speed (HISP) channels - 200 Mbytes/s
- Very high-speed (VHISP) channels - 1800 Mbytes/s

Секция межпроцессорного взаимодействия

Секция межпроцессорного взаимодействия содержит разделяемые регистры и семафоры, предназначенные для передачи данных и управляющей информации между процессорами. Регистры и семафоры разделены на одинаковые группы (кластеры), каждый кластер содержит 8 (32-разрядных) разделяемых адресных (SB) регистра, 8 (64-разрядных) разделяемых скалярных (ST) регистра и 32 однобитовых семафора.

21. Регистровая структура и функциональные устройства процессора CRAY C90.

Все процессоры имеют одинаковую вычислительную секцию, состоящую из регистров, функциональных устройств (ФУ) и сети коммуникаций. Регистры и ФУ могут хранить и обрабатывать три типа данных: адреса (*A*-регистры, *B*-регистры), скаляры (*S*-регистры, *T*-регистры) и вектора (*V*-регистры).

Регистры

Каждый процессор имеет три набора основных регистров (*A*, *S*, *V*), которые имеют связь как с памятью, так и с ФУ. Для регистров *A* и *S* существуют промежуточные наборы регистров *B* и *T*, играющие роль буферов для основных регистров.

Адресные регистры: *A*-регистры, 8 штук по 32 разряда, для хранения и вычисления адресов, индексации, указания величины сдвигов, числа итераций циклов и т.д. *B*-регистры, 64 штуки по 32 разряда.

Скалярные регистры: *S*-регистры, 8 штук по 64 разряда, для хранения аргументов и результатов скалярной арифметики, иногда содержат операнд для векторных команд. *T*-регистры, 64 штуки по 64 разряда. Скалярные регистры используются для выполнения как скалярных, так и векторных команд.

Векторные регистры: *V*-регистры, 8 штук на 128 64-разрядных слова каждый. Векторные регистры используются только для выполнения векторных команд.

Регистр длины вектора: 8 разрядов.

Регистр маски вектора: 128 разрядов.

Функциональные устройства

ФУ исполняют свой набор команд и могут работать одновременно друг с другом. Все ФУ конвейерные и делятся на четыре группы: адресные, скалярные, векторные и для работы с плавающей точкой.

Адресные ФУ (2): целочисленное сложение/вычитание, целочисленное умножение.

Скалярные ФУ (4): целочисленное сложение/вычитание, логические поразрядные операции, сдвиг, число единиц/число нулей до первой единицы.

Векторные ФУ (5-7): целочисленное сложение/вычитание, сдвиг, логические поразрядные операции (1-2), число единиц/число нулей до первой единицы (1-2), умножение битовых матриц (0-1). Предназначены для выполнения только векторных команд.

ФУ с плавающей точкой (3): сложение/вычитание, умножение, нахождение обратной величины. Предназначены для выполнения как векторных, так и скалярных команд.

Векторные ФУ и ФУ с плавающей точкой продублированы: векторные команды разбивают 128 элементов векторных регистров на четные и нечетные, обрабатываемые одновременно двумя конвейерами (pipe 0, pipe 1). Когда завершается выполнение очередной пары операций результаты записываются на соответствующие четные и нечетные позиции выходного регистра. В полностью скалярных операциях, использующих ФУ с плавающей точкой, работает только один конвейер.

ФУ имеют различное число ступеней конвейера, но каждая ступень срабатывает за один такт, поэтому при полной загрузке все ФУ могут выдавать результат каждый такт.

22. Параллелизм в архитектуре компьютера CRAY C90 (6 особенностей архитектуры).

Конвейеризация выполнения команд

Все основные операции, выполняемые процессором: обращения в память, обработка команд и выполнение инструкций являются конвейерными.

Независимость функциональных устройств

Большинство ФУ в CRAY C90 являются независимыми, поэтому несколько операций могут выполняться одновременно. Для операции *A=(B+C)*D*E* порядок выполнения может быть следующим (все аргументы загружены в *S* регистры). Генерируются три инструкции: умножение *D* и *E*, сложение *B* и *C* и умножение результатов двух предыдущих операций. Первые две операции выполняются одновременно, затем третья.

Векторная обработка

Векторная обработка увеличивает скорость и эффективность обработки за счет того, что обработка целого набора (вектора) данных выполняется одной командой. Скорость выполнения операций в векторном режиме приблизительно в 10 раз выше скорости скалярной обработки. Для фрагмента типа

```
Do i = 1, n
   A(i) = B(i)+C(i)
End Do
```

в скалярном режиме потребуется сгенерировать целую последовательность команд: прочитать элемент B(I), прочитать элемент C(I), выполнить сложение, записать результат в A(I), увеличить параметр цикла, проверить условие цикла. В векторном режиме этот фрагмент преобразуется в: загрузить порцию массива B, загрузить порцию массива C (эти две операции будут выполняться со сдвигом в один такт, т.е. практически одновременно), векторное сложение, запись порции массива в память, если размер массивов больше длины векторных регистров, то повторить эту последовательность некоторое число раз.

Перед тем, как векторная операция начнет выдавать результаты, проходит некоторое время (startup), связанное с заполнением конвейера и подкачкой аргументов. Чем больше длина векторов, тем менее заметным оказывается влияние данного начального промежутка времени на все время выполнения программы.

Векторные операции, использующие различные ФУ и регистры, могут выполняться параллельно.

Зацепление функциональных устройств

Архитектура CRAY Y-MP C90 позволяет использовать регистр результатов векторной операции в качестве входного регистра для последующей векторной операции, т.е. выход

сразу подается на вход. Это называется зацеплением векторных операций. Вообще говоря, глубина зацепления может быть любой, например, чтение векторов, выполнение операции сложения, выполнение операции умножения, запись векторов.

Дублирование конвееров. Векторные ФУ и ФУ с плавающей точкой продублированы: векторные команды разбивают 128 элементов векторных регистров на четные и нечетные, обрабатываемые одновременно двумя конвейерами (pipe 0, pipe 1). Когда завершается выполнение очередной пары операций результаты записываются на соответствующие четные и нечетные позиции выходного регистра. В полностью скалярных операциях, использующих ФУ с плавающей точкой, работает только один конвейер.

16 таких процессоров.

Многопроцессорная обработка: multiprogramming, multitasking

Multiprogramming - выполнение нескольких независимых программ на различных процессорах.

Multitasking - выполнение одной программы на нескольких процессорах.

23. Векторизация программ, необходимые условия векторизации, препятствия для векторизации.
Независимость команд
Тип операндов одинаковый
Шаг одинаковый
24. Причины уменьшения производительности компьютера CRAY C90: закон Амдала, секционирование векторных операций, время разгона конвейера.

Анализ факторов, снижающих реальную производительность компьютеров, начнем с обсуждения известного **закона Амдала**. Смысл его сводится к тому, что время работы программы определяется ее самой медленной частью. В самом деле, предположим, что одна половина некоторой программы - это сугубо последовательные вычисления. Тогда вне всякой зависимости от свойств другой половины, которая может идеально векторизоваться либо вообще выполняться мгновенно, ускорения работы всей программы более чем в два раза мы не получим.

Влияние данного фактора надо оценивать с двух сторон. Во-первых, по природе самого алгоритма все множество операций программы Г разбивается на последовательные операции Г1 и операции Г2, исполняемые в векторном режиме. Если доля последовательных операций велика, то программист сразу должен быть готов к тому, что большого ускорения он никакими средствами не получит и, быть может, следует уже на этом этапе подумать об изменении алгоритма.

Во-вторых, не следует сбрасывать со счетов и качество компилятора, который может не распознать векторизуемость отдельных конструкций и, тем самым, часть "потенциально хороших" операций из Г2 перенести в Г1.

Следующие два фактора, снижающие реальную производительность (они же определяют невозможность достижения пиковой производительности) - **секционирование длинных векторных операций** на порции по 128 элементов (векторные регистры имееют 128 64 разр слова) и **время начального разгона конвейера**, относятся к накладным расходам на организацию векторных операций на конвейерных функциональных устройствах. Временем начального разгона конвейера, в частности, определяется тот факт, что очень короткие циклы выгоднее выполнять не в векторном режиме, а в скалярном, когда этих накладных расходов нет.

В отличие от секционирования операций дополнительное время на разгон конвейера требуется лишь один раз при старте векторной операции. Это стимулирует к работе с длинными векторами данных, так как с ростом длины вектора доля накладных расходов в общем времени выполнения операции быстро падает.

25. Причины уменьшения производительности компьютера CRAY C90: конфликты в памяти, ограниченная пропускная способность каналов передачи данных, необходимость использования векторных регистров.

Конфликты при обращении в память у компьютеров CRAY Y-MP полностью определяются аппаратными особенностями организации доступа к оперативной памяти. Память компьютеров CRAY Y-MP C90 в максимальной конфигурации разделена на 8 секций, каждая секция - на 8 подсекций, а каждая подсекция на 16 банков памяти. Ясно, что наибольшего времени на разрешение конфликтов потребуется при выборке данных с шагом 8*8=64, когда постоянно совпадают номера и секций и подсекций. С другой стороны, выборка с любым нечетным шагом проходит без конфликтов вообще, и в этом смысле она эквивалентна выборке с шагом единица. Возьмем следующий пример:

```
Do i=1,n*k,k
  a(i)=b(i)*s + c(i)
End Do
```

В зависимости от значения k, т.е. шага выборки данных из памяти, происходит выполнение векторной операции ai=bi*s+ci длины n в режиме с зацеплением. Производительность компьютера (с двумя секциями памяти) на данной операции показана в таблице 2.

шаг по памяти	производительность на векторах из		
	100 элементов	1000 элементов	12800 элементов
1	240.3	705.2	805.1
2	220.4	444.6	498.5
4	172.9	274.6	280.1
8	108.1	142.8	147.7
16	71.7	84.5	86.0
32	41.0	44.3	38.0
64	22.1	25.7	22.3
128	21.2	20.6	20.3

Табл.2 Влияние конфликтов при обращении к памяти: производительность компьютера CRAY Y-MP C90 в зависимости от длины векторов и шага перемещения по памяти.

Как видим, производительность падает катастрофически. Однако еще одной неприятной стороной конфликтов является то, что "внешних" причин их появления может быть много (в то время как истинная причина, конечно же, одна - неудачное расположение данных). В самом деле, в предыдущем примере конфликты возникали при использовании цикла с нееденичным четным шагом. Значит, казалось бы, не должно быть никаких причин для возникновения конфликтов при работе фрагмента следующего вида:

```
Do i=1,n
Do j=1,n
Do k=1,n
  X(i,j,k) = X(i,j,k)+P(k,i)*Y(k,j)
End Do
End Do
End Do
```

Однако это не совсем так и все зависит от того, каким образом описан массив X. Предположим, что описание имеет вид:

DIMENSION X(40,40,100)

По определению Фортрана массивы хранятся в памяти "по столбцам", следовательно при изменении последнего индексного выражения на единицу реальное смещение по памяти будет равно произведению размеров массива по предыдущим размерностям. Для нашего примера, расстояние между соседними элементами X(i,j,k) и X(i,j,k+1) равно 40*40=1600=25*64, т.е. всегда кратно наихудшему шагу для выборки из памяти. В тоже время, если изменить лишь описание массива, добавив единицу к первым двум размерностям:

DIMENSION X(41,41,100),

то никаких конфликтов не будет вовсе. Последний пример возможного появления конфликтов - это использование косвенной адресации. В следующем фрагменте

```
Do j=1,n
Do i=1,n
  XYZ(IX(i),j) = XYZ(IX(i),j)+P(i,j)*Y(i,j)
End Do
End Do
```

в зависимости от того, к каким элементам массива XYZ реально происходит обращение, конфликтов может не быть вовсе (например, IX(i) равно i) либо их число может быть максимальным (например, IX(i) равно одному и тому же значению для всех i).

Следующие два фактора, снижающие производительность, определяются тем, что перед началом выполнения любой операции данные должны быть занесены в регистры. Для этого в архитектуре компьютера CRAY Y-MP предусмотрены три независимых канала передачи данных, два из которых могут работать на чтение из памяти, а третий на запись. Такая структура хорошо подходит для операций, требующих не более двух входных векторов для выполнения в максимально производительном режиме с зацеплением, например, Ai = Bi*s+Ci.

Однако операции с тремя векторными аргументами, как например Ai = Bi*Ci+Di, не могут быть реализованы столь же оптимально. Часть времени будет неизбежно потрачено впустую на ожидание подкачки третьего аргумента для запуска операции с зацеплением, что является прямым следствием **ограниченной пропускной способности каналов передачи данных** (memory bottleneck). С одной стороны, максимальная производительность достигается на операции с зацеплением, требующей три аргумента, а с другой на чтение одновременно могут работать лишь два канала. В таблице 3 приведена производительность компьютера на указанной выше векторной операции, требующей три входных вектора B, C, D, в зависимости от их длины.

длина вектора	производительность, Mflop/s
10	57.0
100	278.3
1000	435.3
12801	445.0

Табл.3 Производительность CRAY Y-MP C90 на операции ai=bi*ci+di

Теперь предположим, что пропускная способность каналов не является узким местом. В этом случае на предварительное занесение данных в регистры все равно требуется некоторое дополнительное время. Поскольку нам **необходимо использовать векторные регистры** перед выполнением операций, то требуемые для этого операции чтения/записи будут неизбежно снижать общую производительность. Довольно часто влияние данного фактора можно заметно ослабить, если повторно используемые вектора

один раз загрузить в регистры, выполнить построенные на их основе выражения, а уже затем перейти к оставшейся части программы. Рассмотрим следующий фрагмент:

```
Do j=1,120
Do i=1,n
  DP(i) = DP(i) + s*P(i,j-1) + t*P(i,j)
End Do
End Do
```

На каждой итерации по j для выполнения векторной операции требуются три входных вектора DP(i), P(i,j-1), P(i,j) и один выходной - DP(i), следовательно за время работы всего фрагмента будет выполнено 120*3=360 операций чтения векторов и 120 операций записи. Если явно выписать каждые две последовательные итерации цикла по j и преобразовать фрагмент к виду:

```
Do j=1,120,2
Do i=1,n
  DP(i) = DP(i)+s*P(i,j-1)+t*P(i,j)+s*P(i,j)+t*P(i,j+1)
End Do
End Do
```

то на каждой из 60-ти итераций внешнего цикла потребуется четыре входных вектора DP(i), P(i,j-1), P(i,j), P(i,j+1) и опять же один выходной. Суммарно, для нового варианта будет выполнено 60*4=240 операций чтения и 60 операций записи. Преобразование подобного рода носит название "раскрутки" и имеет максимальный эффект в том случае, когда на соседних итерациях цикла используются одни и те же данные.

26. Причины уменьшения производительности компьютера CRAY C90: ограниченный набор векторных регистров, несбалансированность в использовании ФУ, отсутствие устройства деления, перезагрузка буферов команд.

Теоретически, одновременно с увеличением глубины раскрутки растет и производительность, приближаясь в пределе к некоторому значению. Однако на практике максимальный эффект достигается где-то на первых шагах, а затем производительность либо остается примерно одинаковой, либо падает. Основная причина данного несоответствия теории и практики заключается в том, что компьютеры CRAY Y-MP C90 имеют сильно **ограниченный набор векторных регистров**: 8 регистров по 128 слова в каждом. Как правило, любая раскрутка требует подкачки дополнительных векторов, а, следовательно, и дополнительных регистров. Таблица 4 содержит данные по производительности CRAY Y-MP C90 на обсуждаемом выше фрагменте в зависимости от длины векторов и глубины раскрутки.

глубина раскрутки	производительность на векторах из		
	64 элементов	128 элементов	12800 элементов
1	464.4	612.9	749.0
2	591.4	731.6	730.1
3	639.3	780.7	752.5
4	675.3	807.7	786.8

Табл.4 Зависимость производительности компьютера CRAY Y-MP C90 от глубины раскрутки и длины векторов.

Теперь вспомним, что значение пиковой производительности вычислялось при условии одновременной работы всех функциональных устройств. Значит, если некоторый алгоритм выполняет одинаковое число операций сложения и умножения, но все сложения выполняются сначала и лишь затем операции умножения, то в каждый момент времени в компьютере будут задействованы только устройства одного типа. Присутствующая **несбалансированность в использовании функциональных устройств** является

серьезным фактором, сильно снижающим реальную производительность компьютера - соответствующие данные можно найти в таблице 5.

В наборе функциональных устройств **нет устройства деления**. Для выполнения данной операции используется устройство обратной аппроксимации и устройство умножения. Отсюда сразу следует, что, во-первых, производительность фрагмента в терминах операций деления будет очень низкой и, во-вторых, использование деления вместе с операцией сложения немного выгоднее, чем с умножением тк заняты разные ФУ. Конкретные значения производительности показаны в таблице 5.

длина вектора	производительность на операции				
	ai=bi+ci	ai=bi*ci	ai=bi/ci	ai=s/bi+t	ai=s/bi*t
10	35.5	41.9	24.8	45.7	46.1
100	202.9	198.0	88.4	197.4	166.5
1000	343.8	341.2	117.2	283.8	215.9
12800	373.1	376.8	120.0	297.0	222.5

Табл.5 Производительность CRAY Y-MP C90 на операциях одного типа и операциях с делением.

Если структура программы такова, что в ней либо происходит частое обращение к различным небольшим подпрограммам и функциям, либо структура управления очень запутана и построена на основе большого числа переходов, то потребуется частая **перезагрузка буферов команд**, а значит, возникнут дополнительные накладные расходы. Наилучший результат достигается в том случае, если весь фрагмент кода уместился в одном буфере команд. Незначительные потери производительности будут у фрагментов, расположенных в нескольких буферах. Если же перезагрузка частая, т.е. фрагмент или программа обладают малой локальностью вычислений, то производительность может изменяться в очень широких пределах в зависимости от способа организации каждой конкретной программы.

27. Архитектура компьютера NEC Earth Simulator.

Японский компьютер Earth Simulator имеет столь впечатляющие параметры, что его создание стало возможным только в результате выполнения проекта действительно общенационального масштаба. Производство компьютера было закончено в феврале 2002 года, после чего он был установлен в Японском центре морских наук и технологий (Japan Marine Science and Technology Center). Сам компьютер и все его технологическое окружение (системы электропитания, кондиционирования, освещения, сейсмозащиты и т.п.) занимают здание размером 50x65x17 м.

Earth Simulator содержит 640 процессорных узлов, соединенных между собой через высокоскоростной переключатель. В состав узла входят 8 векторных арифметических процессоров, работающих над общей для каждого узла оперативной памятью, коммуникационный процессор и процессор для операций ввода/вывода. Оперативная память каждого узла разделена на 2048 банков и имеет объем 16 Гбайт. Пиковая производительность одного арифметического процессора равна 8 Гфлопс, поэтому пиковая производительность всего компьютера, объединяющего 640*8=5120 процессоров, равна 40 Тфлопс. С лета 2002 года до октября 2003 г. Earth Simulator занимал первое место в списке Top500 самых мощных компьютеров мира, показав производительность 35,86 Тфлопс на тесте Linpack (89,6% от пика).

Скорость двунаправленной передачи по каждому каналу, соединяющему процессорные узлы с переключателем, равна 12.3 Гбайт/с. Суммарная длина кабелей, соединяющих процессорные узлы с переключателем, составляет 2400 км.

Каждый арифметический процессор состоит из скалярного и векторного устройств, а также модуля доступа к общей памяти. Процессор работает на частоте 500 МГц, но

некоторые его компоненты поддерживают работу на частоте 1 ГГц. Скалярное устройство имеет суперскалярную архитектуру, объединяет кэш-память команд и данных по 64 Кбайт каждая и 128 скалярных регистров общего назначения. Каждое векторное устройство имеет 8 наборов, состоящих из 72 векторных регистров (по 256 элементов каждый) и 6 конвейерных устройств: сложение, умножение, деление, для логических операций, операций маскирования и чтения/записи. Устройства одного и того же типа из разных наборов одновременно обрабатывают одну и ту же векторную команду, в то время как устройства различных типов даже в рамках одного набора могут работать одновременно и независимо друг от друга.

Система архивирования компьютера Earth Simulator включает дисковые массивы на 250 Тбайт и ленточную библиотеку StorageTek 9310 на 1.5 Пбайт (1 Петабайт = 10^15 байт).

Архитектура компьютера объединяет многие известные принципы построения высокопроизводительных систем. В целом, Earth Simulator является массивно-параллельным компьютером с распределенной памятью. Вместе с тем, каждый процессорный узел построен на принципах SMP-архитектуры, причем основа каждого процессора - векторно-конвейерная обработка.

Основные технологии параллельного программирования, используемые на компьютере Earth Simulator: Message Passing Interface (MPI), High Performance Fortran, OpenMP и векторизация циклов компилятором.

28. Параллелизм на уровне машинных команд, суперскалярные, VLIW и EPIC архитектуры.

Исключительно интересной оказалась идея воспользоваться скрытым от пользователя параллелизмом — *параллелизмом на уровне машинных команд* (Instruction-Level Parallelism). Выгод — масса, и среди главных стоит назвать отсутствие у пользователя необходимости в специальном параллельном программировании, и то, что проблемы с переносимостью, вообще говоря, остаются на уровне общих проблем переносимости программ в классе последовательных машин.

Существует два основных подхода к построению архитектуры процессоров, использующих параллелизм на уровне машинных команд. В обоих случаях предполагается, что процессор содержит несколько функциональных устройств, которые могут работать независимо друг от друга. Устройства могут быть одинаковыми, могут быть разными — в данном случае это неважно.

Суперскалярные процессоры не предполагают, что программа в терминах машинных команд будет включать в себя какую-либо информацию о содержащемся в ней параллелизме. Задача обнаружения параллелизма в машинном коде возлагается на аппаратуру, она же и строит соответствующую последовательность исполнения команд. В этом смысле код для суперскалярных процессоров не отражает точно ни природу аппаратного обеспечения, на котором он будет реализован, ни точного временного порядка, в котором будут выполняться команды.

VLIW-процессоры (Very Large Instruction Word) работают практически по правилам фоннеймановского компьютера. Разница лишь в том, что команда, выдаваемая процессору на каждом цикле, определяет не одну операцию, а сразу несколько. Команда VLIW-процессора состоит из набора полей, каждое из которых отвечает за свою операцию, например, за активизацию функциональных устройств, работу с памятью, операции с регистрами и т. п. Если какая-то часть процессора на данном этапе выполнения программы не востребована, то соответствующее поле команды не задействуется.

Примером компьютера с подобной архитектурой может служить компьютер AP-120B фирмы Floating Point Systems. Его первые поставки начались в 1976 году, а к 1980 году по всему миру было установлено более 1600 экземпляров. Команда компьютера AP-

120B состоит из 64 разрядов и управляет работой всех устройств машины. Каждый такт (167 не) выдается одна команда, что эквивалентно выполнению 6 миллионов команд в секунду. Поскольку каждая команда одновременно управляет многими операциями, то реальная производительность может быть выше. Все 64 разряда команды AP-120B делятся на шесть групп, отвечающих за свой набор операций: операции над 16-разрядными целочисленными данными и регистрами, сложение вещественных чисел, управление вводом/выводом, команды перехода, умножение вещественных чисел и команды работы с основной памятью.

Программа для VLIW-процессора всегда содержит точную информацию о параллелизме. Здесь компилятор всегда сам выявляет параллелизм в программе и явно сообщает аппаратуре, какие операции не зависят друг от друга. Код для VLIW-процессоров содержит точный план того, как процессор будет выполнять программу: когда будет выполнена каждая операция, какие функциональные устройства будут работать, какие регистры какие операнды будут содержать и т. п. Компилятор VLIW создает такой план имея полное представление о целевом VLIW-процессоре, чего, вообще говоря, нельзя сказать о компиляторах для суперскалярных машин.

Оба подхода имеют свои достоинства и недостатки, и не стоит противопоставлять простоту и ограниченные возможности архитектуры VLIW сложности и динамическим возможностям суперскалярных систем. Ясно, что создание плана выполнения операций во время компиляции важно для обеспечения высокой степени распараллеливания и в отношении суперскалярных систем. Также ясно и то, что во время компиляции существует неоднозначность, которую можно разрешить только во время выполнения программы с помощью динамических механизмов, свойственных суперскалярной архитектуре.

Большое влияние на развитие идей суперскалярной обработки еще в конце 50-х годов прошлого столетия оказал проект STRETCH фирмы IBM, и сейчас архитектура многих микропроцессоров построена по этому же принципу. Яркими представителями VLIW-компьютеров являются компьютеры семейств Multiflow и Cydra.

29. Технологии параллельного программирования: способы и подходы

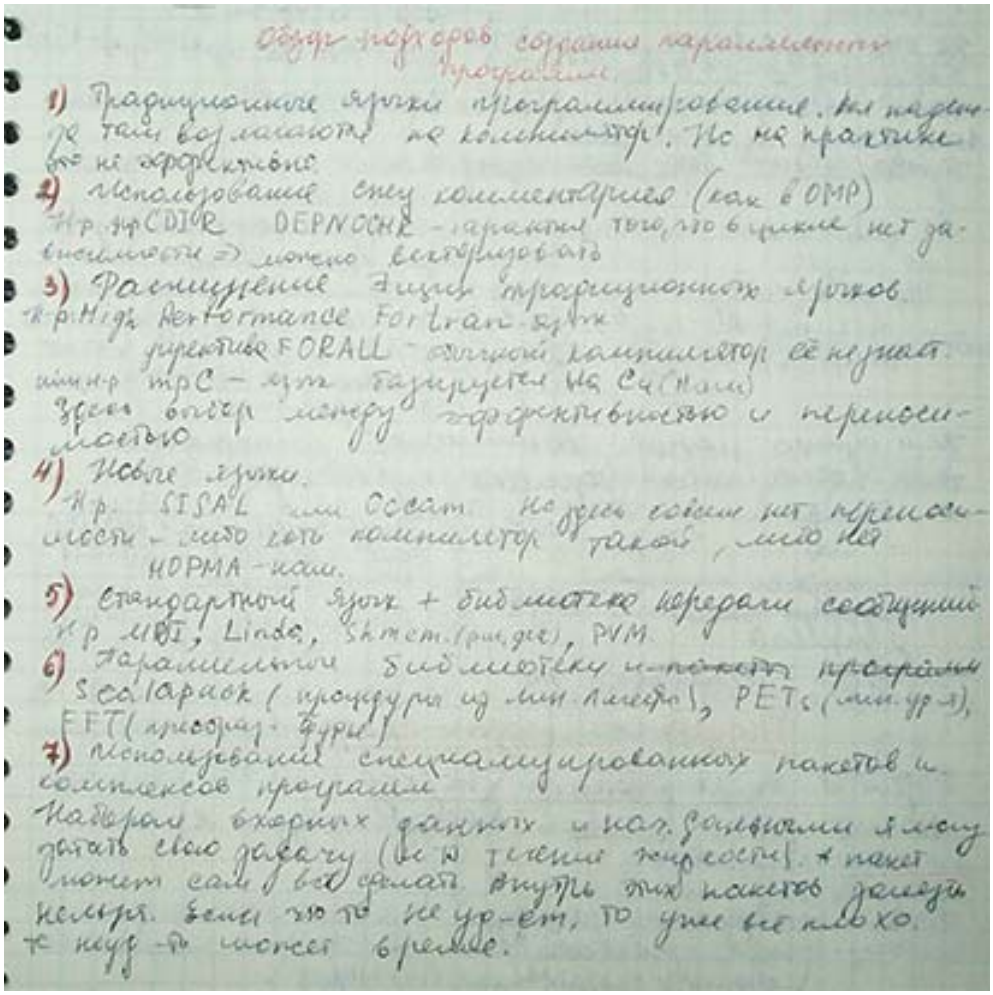
Технологии параллельного программирования

- возможность создания переносимых параллельных программ
- возможность быстро создания программ
- переносимость программ
- стоимость

0. Распараллеливающий компилятор
1. Стандартизация
2. Расширения существующих языков программирования
3. Специальные языки программирования
   - Occam - для транспьютеров
   - НОРМА, ЯПМРАН

   декларативные языки программирования
4. Библиотеки и интерфейсы
   - MPI
   - PVM
   - Shmem

5. Параллельные предм. библиотеки
   - PBLAS
   - ScaLAPACK
   - ATLAS
   - MKL
   - FFTW, ⊗FFTPack
   - PETSc
6. Спец. пакеты пр-м
   - GAMESS



30. Linda: общая концепция, пространство кортежей, примеры программ.

Идея построения системы Linda исключительно проста, а потому красива и очень привлекательна. Параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно обычной последовательной программе. Все процессы имеют доступ к общей памяти, единицей хранения в которой является кортеж. Отсюда происходит и специальное название для общей памяти - пространство кортежей. Каждый кортеж это упорядоченная последовательность значений. Например, ("Hello", 42, 3.14). Количество элементов в кортеже может быть любым.

Все процессы работают с пространством кортежей по принципу: поместить кортеж, забрать, скопировать. В отличие от традиционной памяти, процесс может забрать кортеж из пространства кортежей, после чего данный кортеж станет недоступным остальным процессам. В отличие от традиционной памяти, если в пространство кортежей положить два кортежа с одним и тем же именем, то не произойдет привычного для нас "обновления" значения переменной - в пространстве кортежей окажется два кортежа с одним и тем же именем. В отличие от традиционной памяти, изменить кортеж непосредственно в пространстве нельзя. Для изменения значений элементов кортежа, его нужно сначала оттуда изъять, затем процесс, изъявший кортеж, может изменить значения его элементов и вновь добавить измененный кортеж в память. В отличие от других систем программирования, процессы в системе Linda никогда не взаимодействуют друг с другом явно, и все общение всегда идет через пространство кортежей.

Интересно, что с точки зрения системы Linda в любой последовательный язык достаточно добавить лишь четыре новые функции, как он становится средством параллельного программирования! Эти функции и составляют систему Linda: три для операций над кортежами и пространством кортежей и одна функция для порождения параллельных процессов.

Пример 1.

По сути дела, описание системы закончено, и теперь можно привести несколько небольших примеров. Мы уже говорили о том, что параллельные процессы в системе Linda напрямую друг с другом не общаются, своего уникального номера-идентификатора не имеют и общего числа параллельно работающих процессов-соседей, вообще говоря, не знают. Однако если у пользователя есть в этом необходимость, то такую ситуацию очень просто смоделировать. Программа в самом начале вызывает функцию out:

```
out( "Next", 1);
```

Этот кортеж будет играть роль "эстафетной палочки", передаваемой от процесса процессу: каждый порождаемый параллельный процесс первым делом выполнит следующую последовательность:

```
in( "Next", formal My_Id);
out( "Next", My_Id+1);
```

Первый оператор изымает данный кортеж из пространства, на его основе процесс получает свой номер My_Id, и затем кортеж с номером для следующего процесса помещается в пространство. Заметим, что использование функции in в данном случае позволяет гарантировать монопольную работу с данным кортежем только одного процесса в каждый момент времени. После такой процедуры каждый процесс получит свой уникальный номер, а число уже порожденных процессов всегда можно определить, например, с помощью такого оператора:

```
read( "Next", formal Num_Processes);
```

Пример 2.

Теперь рассмотрим возможную схему организации программы для перемножения C=A*B двух квадратных матриц размера N*N. Инициализирующий процесс использует функцию out и помещает в пространство кортежей исходные строки матрицы A и столбцы матрицы B:

```
out( "A",1, <1-я строка A>);
out( "A",2, <2-я строка A>);
        ...
out( "B",1, <1-й столбец B>);
out( "B",2, <2-й столбец B>);
        ...
```

Для порождения Nproc идентичных параллельных процессов можно воспользоваться следующим фрагментом:

```
for( i = 0; i < Nproc; ++i )
        eval( "ParProc", get_elem_result() );
```

Входные данные готовы, и нахождение всех N2 элементов Cij результирующей матрицы можно выполнять в любом порядке. Главное - это распределить работу между процессами, для чего процесс, инициирующий вычисления, в пространство помещает следующий кортеж:

```
out( "NextElementCij", 1);
```

Второй элемент данного кортежа всегда будет показывать, какой из N2 элементов Cij предстоит вычислить следующим. Базовый вычислительный блок функции get_elem_result() будет содержать следующий фрагмент:

```
in( "NextElementCij", formal NextElement);
if( NextElement < N*N )
        out("NextElementCij ", NextElement + 1);
Nrow = (NextElement - 1) / N + 1;
Ncol = (NextElement - 1) % N + 1;
```

В результате выполнения данного фрагмента для элемента с номером NextElement процесс определит его местоположение в результирующей матрице: номер строки Nrow и столбца Ncol. Заметим, что если вычисляется последний элемент, то кортеж с именем "NextElementCij" в пространство не возвращается. Когда в конце работы программы процессы обратятся к этому кортежу, они будут заблокированы, что не помешает нормальному завершению программы. И, наконец, для вычисления элемента Cij каждый процесс get_elem_result выполнит следующий фрагмент:

```
read( "A", Nrow, formal row);
read( "B", Ncol, formal col);
out( "result", Nrow, Ncol, DotProduct(row,col) );
```

где DotProduct это функция, реализующая скалярное произведение. Таким образом, каждый элемент произведения окажется в отдельном кортеже в пространстве кортежей. Завершающий процесс соберет результат, поместив их в соответствующие элементы матрицы C:

```
for ( irow = 0; irow < N; irow++)
        for ( icol = 0; icol < N; icol++)
                in( "result", irow + 1, icol + 1, formal C[irow][icol]);
```

Пример 3.

Не имея в системе Linda никаких явных средств для синхронизации процессов, совсем не сложно их смоделировать самому. Предположим, что в некоторой точке нужно выполнить барьерную синхронизацию N процессов. Какой-то один процесс, например, стартовый, заранее помещает в пространство кортеж ("ForBarrier", N). Подходя к точке синхронизации, каждый процесс выполняет следующий фрагмент, который и будет выполнять функции барьера:

```
in( "ForBarrier", formal Bar);
Bar = Bar - 1;
if( Bar != 0 ) {
        out( "ForBarrier", Bar);
        read( "Barrier" );
} else
        out( "Barrier" );
```

Если кортеж с именем "ForBarrier" есть в пространстве, то процесс его изымает, в противном случае блокируется до его появления. Анализируя второй элемент данного кортежа, процесс выполняет одно из двух действий. Если есть процессы, которые еще не дошли до данной точки, то он возвращает кортеж в пространство с уменьшенным на единицу вторым элементом и встает на ожидание кортежа "Barrier". В противном случае он сам помещает кортеж "Barrier" в пространство, который для всех является сигналом к продолжению работы.

31. Linda: специальные функции для работы с пространством кортежей.

Функция **OUT** помещает кортеж в пространство кортежей. Например, *out ("GoProcess", 5);* помещает в пространство кортежей кортеж ("GoProcess", 5). Если такой кортеж уже есть в пространстве кортежей, то появится второй, что, в принципе, позволяет иметь сколь угодно много экземпляров одинаковых кортежей. По этой же причине с помощью функции out нельзя изменить кортеж, уже находящийся в пространстве. Для этого кортеж должен быть сначала оттуда изъят, затем изменен и после этого помещен назад. Функция out никогда не блокирует выполнивший ее процесс.

Функция **IN** ищет подходящий кортеж в пространстве кортежей, присваивает значения его элементов элементам своего параметра-кортежа и удаляет найденный кортеж из пространства кортежей. Например,

```
in( "P", int i, FALSE );
```

Этой функции соответствует любой кортеж, который состоит из трех элементов: значением первого элемента является "P", второй элемент может быть любым целым числом, а третий должен иметь значение FALSE. Подходящими кортежами могут быть ("P", 5, FALSE) или ("P", 135, FALSE) и т.п., но не ("P", 7.2, FALSE) или ("Proc", 5, FALSE). Если параметру функции in соответствуют несколько кортежей, то случайным образом выбирается один из них. После нахождения кортеж удаляется из пространства кортежей, а неопределенным формальным элементам параметра-кортежа, содержащимся в вызове данной функции, присваиваются соответствующие значения. В предыдущем примере переменной i присвоится 5 или 135. Если в пространстве кортежей ни один кортеж не соответствуют функции, то вызвавший ее процесс блокируется до тех пор, пока соответствующий кортеж в пространстве не появится.

Элемент кортежа в функции in считается формальным, если перед ним стоит определитель типа. Если используется переменная без определителя типа, то берется ее значение и элемент рассматривается как фактический параметр. Например, во фрагменте программы

int i = 5;
in("P", i, FALSE);

функции in, в отличие от предыдущего примера, соответствует только кортеж ("P", 5, FALSE).

Если переменная описана до вызова функции и ее надо использовать как формальный элемент кортежа, можно использовать ключевое слово formal или знак '?'. Например, во фрагменте программы

j = 15;
in("P", formal i, j);

последнюю строку можно заменить и на оператор in("P", ?i, j). В этом примере функции in будет соответствовать, например, кортеж ("P", 6, 15), но не ("P", 6, 12). Конечно же, формальными могут быть и несколько элементов кортежа одновременно:

in ("Add_If", int i, bool b);

Если после такого вызова функции в пространстве кортежей будет найден кортеж ("Add_If", 100, TRUE), то переменной i присвоится значение 100, а переменной b - значение TRUE.

Функция **READ** отличается от функции in лишь тем, что выбранный кортеж не удаляется из пространства кортежей. Все остальное точно так же, как и у функции in. Этой функцией удобно пользоваться в том случае, когда значения переменных менять не нужно, но к ним необходим параллельный доступ из нескольких процессов.

Функция **EVAL** похожа на функцию out. Разница заключается лишь в том, что дополнительным элементом кортежа у eval является функция пользователя. Для вычисления значения этой функции система Linda порождает параллельный процесс, на основе работы которого она формирует кортеж и помещает его в пространство кортежей. Например,

eval ("hello", funct(z), TRUE, 3.1415);

При обработке данного вызова система создаст новый процесс для вычисления функции funct(z). Когда процесс закончится и будет получено значение w = funct(z), в пространство кортежей будет добавлен кортеж ("hello", w, TRUE, 3.1415). Функция, вызвавшая eval, не ожидает завершения порожденного параллельного процесса и продолжает свою работу дальше. Следует отметить и то, что пользователь не может явно управлять размещением порожденных параллельных процессов на доступных ему процессорных устройствах - это Linda делает самостоятельно.

Параллельная программа в системе Linda считается завершенной, если все порожденные процессы завершились или все они заблокированы функциями in и read.

32. MPI: общая структура. Основные отличия MPI-2 от MPI-1.

Наиболее распространенной технологией программирования параллельных компьютеров с распределенной памятью в настоящее время является MPI. Мы уже говорили о том, что основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии данной технологии — Message Passing Interface. Стандарт MPI фиксирует интерфейс, который должны соблюдать как система программирования MPI на каждой вычислительной системе, так и пользователь при создании своих программ. Современные реализации чаще всего соответствуют стандарту MPI версии 1.1. В 1997—1998 годах появился стандарт MPI-2.0, значительно расширивший функциональность предыдущей версии. Однако до сих пор этот вариант MPI не получил широкого распространения. Везде далее, если иного не оговорено, мы будем иметь дело со стандартом 1.1.

MPI поддерживает работу с языками C и Fortran. В данной книге все примеры и описания всех функций будут даны с использованием языка C. Однако это совершенно не является принципиальным, поскольку основные идеи MPI и правила оформления отдельных конструкций для этих языков во многом схожи.

Полная версия интерфейса содержит описание более 120 функций.

Интерфейс поддерживает создание параллельных программ в стиле MIMD, что подразумевает объединение процессов с различными исходными текстами. Однако на практике программисты гораздо чаще используют SPMD-модель, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Все дополнительные объекты: имена функций, константы, предопределенные типы данных и т. п., используемые в MPI, имеют префикс MPI_. Например, функция посылки сообщения от одного процесса другому имеет имя MPi_Send. Если пользователь не будет использовать в программе имен с таким префиксом, то конфликтов с объектами MPI заведомо не будет. Все описания интерфейса MPI собраны в файле mpi.h, поэтому в начале MPI-программы должна стоять директива #include <mpi.h>.

MPI-программа — это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается.

Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Для локализации взаимодействия параллельных процессов программы можно создавать группы процессов, предоставляя им отдельную среду для общения — коммуникатор. Состав образуемых групп произволен. Группы могут полностью входить одна в другую, не пересекаться или пересекаться частично. При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя MPI_COMM_WORLD. Этот коммуникатор существует всегда и служит для взаимодействия всех процессов MPI-программы.

Каждый процесс MPI-программы имеет уникальный атрибут номер процесса, который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда два основных атрибута процесса: коммуникатор и номер в коммуникаторе.

Если группа содержит n процессов, то номер любого процесса в данной группе лежит в пределах от 0 до n - 1. Подобная линейная нумерация не всегда адекватно отражает логическую взаимосвязь процессов приложения. Например, по смыслу задачи процессы могут располагаться в узлах прямоугольной решетки и взаимодействовать

только со своими непосредственными соседями. Такую ситуацию пользователь может легко отразить в своей программе, описав соответствующую виртуальную топологию процессов. Эта информация может оказаться полезной при отображении процессов программы на физические процессоры вычислительной системы. Сам процесс отображения в MPI никак не специфицируется, однако система поддержки MPI в ряде случаев может значительно уменьшить коммуникационные накладные расходы, если воспользуется знанием виртуальной топологии.

Основным способом общения процессов между собой является посылка сообщений. Сообщение — это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и др. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до 32 767. Для работы с атрибутами сообщений введена структура MPistatus, поля которой дают доступ к значениям атрибутов.

На практике сообщение чаще всего является набором однотипных данных, расположенных подряд друг за другом в некотором буфере. Такое сообщение может состоять, например, из двухсот целых чисел, которые пользователь разместил в соответствующем целочисленном векторе. Это типичная ситуация, на нее ориентировано большинство функций MPI, однако такая ситуация имеет, по крайней мере, два ограничения. Во-первых, иногда необходимо составить сообщение из разнотипных данных. Конечно же, можно отдельным сообщением послать количество вещественных чисел, содержащихся в последующем сообщении, но это может быть и неудобно программисту, и не столь эффективно. Во-вторых, не всегда посылаемые данные занимают непрерывную область в памяти. Если в Fortran элементы столбцов матрицы расположены в памяти друг за другом, то элементы строк уже идут с некоторым шагом. Чтобы послать строку, данные нужно сначала упаковать, передать, а затем вновь распаковать.

Чтобы снять указанные ограничения, в MPI предусмотрен механизм для введения производных типов данных (derived datatypes). Описав состав и схему размещения в памяти посылаемых данных, пользователь в дальнейшем работает с такими типами так же, как и со стандартными типами данных MPI.
Поскольку собственные типы данных и виртуальные топологии процессов используются на практике не очень часто, то в данной книге мы не будем их описывать подробно.

33. MPI: синхронное и асинхронное взаимодействие процессов.
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, а 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.

34. MPI: различные виды операторов Send.

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_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_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_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.

35. MPI: коллективные операции.

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.

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.

36. MPI: группы, коммуникаторы.

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*.

37. Модели параллельных программ: SPMD, мастер/рабочие.

Схема мастер/рабочие (master/slaves). Процесс-мастер порождает какое-то число одинаковых процессов-рабочих, распределяет между ними работу и собирает результат.

Приведём для примера текст подобной программы написанной для системы Linda.

```
main(argc, argv)
int argc;
char *argv[];
{
        int nworker, j, hello();
        nworker = atoi (argv[l]);
        for (j = 0; j < nworker; j++)
                eval ("worker", hello(j));
        for(j =0; j < nworker; j++)
```

```
                in("done");
        printf("Hello_world is finished.\n");
}

int hello (num)          /** function hello **/
int num;
{
        printf("Hello world from number %d.\n", num);
        out("done");
        return(0);
}
```

Если много рабочих то мастер не сможет их всех обслужить.

SPMD-модель. В рамках этой модели для всех параллельных процессов используется один и тот же код. Одна программ много данных.

38. Модели передачи сообщений Send/Recv и Put/Get; свойства программ, написанных в терминах Put/Get.

39. OpenMP: общая концепция.

Одним из наиболее популярных средств программирования компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP. За основу берется последовательная программа, а для создания ее параллельной версии пользователю предоставляется набор директив, процедур и переменных окружения. Стандарт OpenMP разработан для языков Фортран, C и C++. Поскольку все основные конструкции для этих языков похожи, то рассказ о данной технологии мы будем вести на примере только одного из них, а именно на примере языка Фортран.

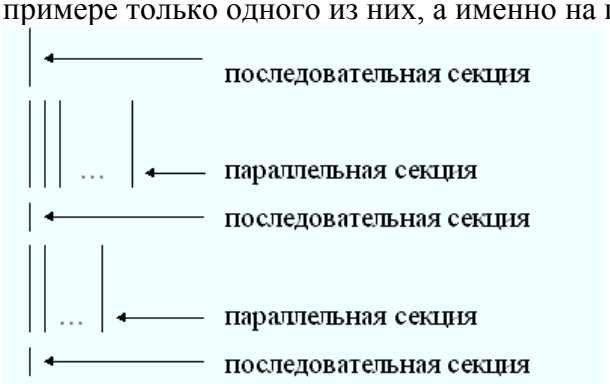


Рис. 1. Процесс исполнения OpenMP-программы

Как, с точки зрения OpenMP, пользователь должен представлять свою параллельную программу? Весь текст программы разбит на последовательные и параллельные области (см. рис.1). В начальный момент времени порождается нить-мастер или "основная" нить, которая начинает выполнение программы со стартовой точки. Здесь следует сразу сказать, почему вместо традиционных для параллельного программирования процессов появился новый термин - нити (threads, легковесные процессы). Технология OpenMP опирается на понятие общей памяти, поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что позволяет избежать значительных накладных расходов на поддержку классических UNIX-процессов.

Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити. После

порождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она.

В параллельной области все переменные программы разделяются на два класса: общие (SHARED) и локальные (PRIVATE). Общая переменная всегда существует лишь в одном экземпляре для всей программы и доступна всем нитям под одним и тем же именем. Объявление же локальной переменной вызывает порождение своего экземпляра данной переменной для каждой нити. Изменение нитью значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других нитях.

По сути, только что рассмотренные два понятия: области и классы переменных, и определяют идею написания параллельной программы в рамках OpenMP: некоторые фрагменты текста программы объявляется параллельными областями; именно эти области и только они исполняются набором нитей, которые могут работать как с общими, так и с локальными переменными. Все остальное - это конкретизация деталей и описание особенностей реализации данной идеи на практике.

40. OpenMP: основные конструкции для организации параллельных и последовательных секций, для распределения работы между нитями.

Описание параллельных областей. Для определения параллельных областей программы используется пара директив

!$OMP PARALLEL

< параллельный код программы >

!$OMP END PARALLEL

Для выполнения кода, расположенного между данными директивами, дополнительно порождается OMP_NUM_THREADS-1 нитей, где OMP_NUM_THREADS - это переменная окружения, значение которой пользователь, вообще говоря, может изменять. Процесс, выполнивший данную директиву (нить-мастер), всегда получает номер 0. Все нити исполняют код, заключенный между данными директивами. После END PARALLEL автоматически происходит неявная синхронизация всех нитей, и как только все нити доходят до этой точки, нить-мастер продолжает выполнение последующей части программы, а остальные нити уничтожаются.

Параллельные секции могут быть вложенными одна в другую. По умолчанию вложенная параллельная секция исполняется одной нитью. Необходимую стратегию обработки вложенных секций определяет переменная OMP_NESTED, значение которой можно изменить с помощью функции OMP_SET_NESTED.

Если значение переменной OMP_DYNAMIC установлено в 1, то с помощью функции OMP_SET_NUM_THREADS пользователь может изменить значение переменной OMP_NUM_THREADS, а значит и число порождаемых при входе в параллельную секцию нитей. Значение переменной OMP_DYNAMIC контролируется функцией OMP_SET_DYNAMIC.

Необходимость порождения нитей и параллельного исполнения кода параллельной секции пользователь может определять динамически с помощью дополнительной опции IF в директиве:

!$OMP PARALLEL IF(<условие>)

Если <условие> не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме.

Мы уже говорили о том, что все порожденные нити исполняют один и тот же код. Теперь нужно обсудить вопрос, как разумным образом распределить между ними работу. OpenMP предлагает несколько вариантов. Можно программировать на самом низком уровне, распределяя работу с помощью функций OMP_GET_THREAD_NUM и

OMP_GET_NUM_THREADS, возвращающих номер нити и общее количество порожденных нитей соответственно. Например, если написать фрагмент вида:

IF(OMP_GET_THREAD_NUM() .EQ. 3) THEN

< код для нити с номером 3 >

ELSE

< код для всех остальных нитей >

ENDIF ,

то часть программы между директивами IF:ELSE будет выполнена только нитью с номером 3, а часть между ELSE:ENDIF - всеми остальными. Как и прежде, этот код будет выполнен всеми нитями, однако функция OMP_GET_THREAD_NUM() возвратит значение 3 только для нити с номером 3, поэтому и выполнение данного участка кода для третьей нити и всех остальных будет идти по-разному.

Если в параллельной секции встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями, т.е. каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву

!$OMP DO [опция [[,] опция]:]

!$OMP END DO ,

которая относится к идущему следом за данной директивой оператору DO.

Опция SCHEDULE определяет конкретный способ распределения итераций данного цикла по нитям:

STATIC [,m] - блочно-циклическое распределение итераций: первый блок из m итераций выполняет первая нить, второй блок - вторая и т.д. до последней нити, затем распределение снова начинается с первой нити; по умолчанию значение m равно 1;

DYNAMIC [,m] - динамическое распределение итераций с фиксированным размером блока: сначала все нити получают порции из m итераций, а затем каждая нить, заканчивающая свою работу, получает следующую порцию опять-таки из m итераций;

GUIDED [,m] - динамическое распределение итераций блоками уменьшающегося размера; аналогично распределению DYNAMIC, но размер выделяемых блоков все время уменьшается, что в ряде случаев позволяет аккуратнее сбалансировать загрузку нитей;

RUNTIME - способ распределения итераций цикла выбирается во время работы программы в зависимости от значения переменной OMP_SCHEDULE.

Выбранный способ распределения итераций указывается в скобках после опции SCHEDULE, например:

!$OMP DO SCHEDULE (DYNAMIC, 10)

В данном примере будет использоваться динамическое распределение итераций блоками по 10 итераций.

В конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки. Если в подобной задержке нет необходимости, то директива END DO NOWAIT позволяет нитям уже дошедшим до конца цикла продолжить выполнение без синхронизации с остальными. Если директива END DO в явном виде и не указана, то в конце параллельного цикла синхронизация все равно будет выполнена.

Параллелизм на уровне независимых фрагментов оформляется в OpenMP с помощью директивы SECTIONS : END SECTIONS:

!$OMP SECTIONS

< фрагмент 1>

!$OMP SECTIONS

< фрагмент 2>

!$OMP SECTIONS

< фрагмент 3>

!$OMP END SECTIONS

В данном примере программист описал, что все три фрагмента информационно независимы, и их можно исполнять в любом порядке, в частности, параллельно друг другу. Каждый из таких фрагментов будет выполнен какой-либо одной нитью.

Если в параллельной секции какой-то участок кода должен быть выполнен лишь один раз (такая ситуация иногда возникает, например, при работе с общими переменными), то его нужно поставить между директивами SINGLE : END SINGLE. Такой участок кода будет выполнен нитью, первой дошедшей до данной точки программы.

41. OpenMP: основные конструкции для синхронизации нитей и работы с общими и локальными данными.

Одно из базовых понятий OpenMP - классы переменных. Все переменные, используемые в параллельной секции, могут быть либо общими, либо локальными. Общие переменные описываются директивой SHARED, а локальные директивой PRIVATE. Каждая общая переменная существует лишь в одном экземпляре и доступна для каждой нити под одним и тем же именем. Для каждой локальной переменной в каждой нити существует отдельный экземпляр данной переменной, доступный только этой нити. Предположим, что следующий фрагмент расположен в параллельной секции:

I = OMP_GET_THREAD_NUM()
*PRINT *, I*

Если переменная I в данной параллельной секции была описана как локальная, то на выходе будет получен весь набор чисел от 0 до OMP_NUM_THREADS-1, идущих, вообще говоря, в произвольном порядке, но каждое число встретиться только один раз. Если же переменная I была объявлена общей, то единственное, что можно сказать с уверенностью - мы получим последовательность из OMP_NUM_THREADS чисел, лежащих в диапазоне от 0 до OMP_NUM_THREADS-1 каждое. Сколько и каких именно чисел будет в последовательности заранее сказать нельзя. В предельном случае, это может быть даже набор из OMP_NUM_THREADS одинаковых чисел I0. Предположим, что все процессы, кроме процесса I0, выполнили первый оператор, но затем их выполнение по какой-то причине было прервано. В это время процесс с номером I0 присвоил это значение переменной I, а поскольку данная переменная является общей, то одно и тоже значение затем и будет выведено каждой нитью.

Целый набор директив в OpenMP предназначен для синхронизации работы нитей. Самый распространенный способ синхронизации - барьер. Он оформляется с помощью директивы

!$OMP BARRIER .

Все нити, дойдя до этой директивы, останавливаются и ждут пока все нити не дойдут до этой точки программы, после чего все нити продолжают работать дальше. Пара директив MASTER : END MASTER выделяет участок кода, который будет выполнен только нитью-мастером. Остальные нити пропускают данный участок и продолжают работу с выполнения оператора, расположенного следом за директивой END MASTER.

С помощью директив

!$OMP CRITICAL [(<имя_критической_секции>)]
...
!$OMP END CRITICAL [(< имя_ критической_секции >)],

оформляется критическая секция программы. В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью P0, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока нить P0 не закончит выполнение данной критической секции. Как только P0 выполнит директиву END CRITICAL, одна из заблокированных на входе нитей войдет в секцию. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити

продолжают ожидание. Все неименованные критические секции условно ассоциируются с одним и тем же именем.

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная SUM является общей и оператор вида SUM=SUM+Expr находится в параллельной секции программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой ATOMIC:

!$OMP ATOMIC
SUM=SUM+Expr .

Данная директива относится к идущему непосредственно за ней оператору, гарантируя корректную работу с общей переменной, стоящей в левой части оператора присваивания.

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени каждая нить будет видеть единый согласованный образ памяти. Именно для этих целей и предназначена директива

!$OMP FLUSH [список_переменных].

Выполнение данной директивы предполагает, что значения всех переменных, временно хранящиеся в регистрах, будут занесены в основную память, все изменения переменных, сделанные нитями во время их работы, станут видимы остальным нитям, если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. Поскольку выполнение данной директивы в полном объеме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в директиве списком.

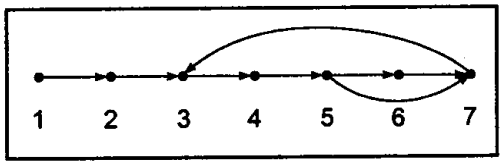
42. Графовые модели программ, их взаимосвязь. Граф алгоритма. Критический путь графа алгоритма.

Рассмотрим 4 основные Графовые модели программ: граф управления программы (ГУ), информационный граф программы (ИГ), операционная история (ОИ), информационная история (ИИ).

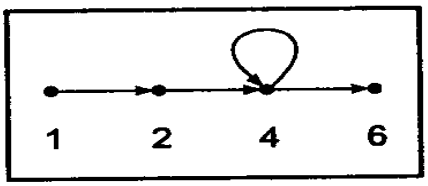
Пример:

```
1 y=b1-a1
2 x=y
3 DO i = 2, n
4       x = (bi – ci*x)/ai
5       if (x<y) goto 7
6       y=x
7 END  DO
```

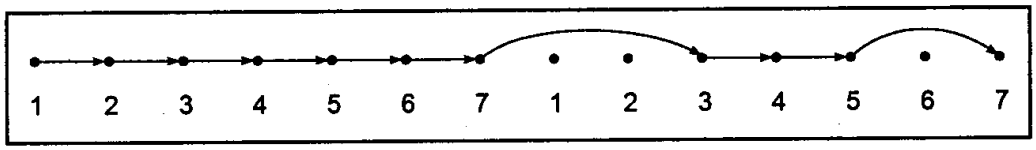
Граф управления. Каждому оператору исходной программы поставим в соответствие вершину графа — экземпляр преобразователя или распознавателя в зависимости от типа оператора. Получим множество вершин, между которыми согласно исходной программе определим отношение, соответствующее передаче управления. Если текст программы допускает выполнение одного оператора непосредственно за другим, то соответствующие вершины соединим дугой, направленной от предшественника к последователю. Его основным свойством является независимость от входных данных программы. Множества вершин и дуг для каждой программы фиксированы и образуют единственный граф.



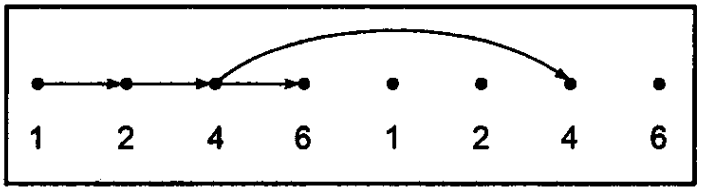
Информационный граф. Изменим графовую основу. Будем среди операторов принимать во внимание только преобразователи, а в качестве отношения между ними брать отношение информационной зависимости. Построим сначала граф, в котором вершины соответствуют операторам-преобразователям. Две вершины соединим информационной дугой, если между какими-нибудь срабатываниями соответствующих операторов теоретически возможна информационная связь. Информационный граф не зависит от входных данных. В нем могут быть "лишние" дуги, которые не реализуются либо при конкретных входных данных, либо совместно с другими дугами.



Операционная история. Теперь предположим, что каким-то образом мы определили начальные данные программы и наблюдаем за ее выполнением на обычном последовательном вычислителе. Каждое срабатывание каждого оператора (а оно необязательно будет единственным) будем фиксировать отдельной вершиной. Получим множество, которое количественно почти всегда будет отличаться от множества вершин графа управления. В данном случае порядок непосредственного срабатывания операторов можно определить точно. Соединяем вершины дугами передач управления, получаем ориентированный граф. Этот граф представляет единственный путь от начальной вершины к конечной. По существу это не что иное, как последовательность срабатывания преобразователей и распознавателей исходной программы при заданных входных данных. В операционной истории от входных данных зависит практически все: общее число вершин, количество вершин, соответствующих одному оператору, и даже набор присутствующих преобразователей и распознавателей. Для графа управления.



Информационная история. Снова каким-то образом определим начальные данные программы и будем наблюдать за ее выполнением на последовательном вычислителе. Каждое срабатывание каждого оператора-преобразователя будем фиксировать отдельной вершиной. Соединим вершины дугами передач информации, получим ориентированный граф. Для информационного графа.



Пусть при фиксированных входных данных программа описывает некоторый алгоритм. Построим ориентированный граф. В качестве вершин возьмем любое множество, например, множество точек арифметического пространства, на которое взаимнооднозначно отображается множество всех операций алгоритма. Возьмем любую пару вершин *u,* v. Допустим, что согласно описанному выше частичному порядку операция, соответствующая вершине *u,* должна поставлять аргумент операции,

соответствующей вершине v. Тогда проведем дугу из вершины *u* в вершину *y*. Если соответствующие операции могут выполняться независимо друг от друга, дугу проводить не будем. В случае, когда аргументом операции является начальное данное или результат операции нигде не используется, возможны различные договоренности. Например, можно считать, что соответствующие дуги отсутствуют. Мы будем поступать в зависимости от обстоятельств. Построенный таким образом граф будем называть *графом алгоритма.* Независимо от способа построения ориентированного графа, те его вершины, которые не имеют ни одной входящей или выходящей дуги, будем называть соответственно *входными* или *выходными* вершинами графа.

43. Пространство итераций, стандартная линейная форма, линейный класс программ.

Для нормализованного цикла

for (i=1;…)

for(j=1;…)

for(k=1;…) T(i,j,k) – тело цикла

Пространство итераций ⟶

Стандартная линейная форма (a,I) + (b,N) + b0

Исследуем линейные многогранники: $(a,I)+b_0<=0$, где I=(i,j,k)

Параметры цикла – переменные n_1, n_2 - внешние переменные

$(a,I)+(b,N)+b_0<=0$

$(a,I)+(b,N)+b_0$ - стандартная линейная форма

Программа относится к **линейному классу программ**, если удовлетворяет следующим условиям:

* в программе может использоваться любое число простых переменных и переменных с индексами;
* единственным типом исполнительного оператора может быть оператор присваивания, правая часть которого есть арифметическое выражение; допускается любое число таких операторов;
* все повторяющиеся операции описываются только с помощью стандартного цикла for;
* допускается использование любого числа условных и безусловных операторов перехода, передающих управление "вниз" по тексту; не допускается использование побочных выходов из циклов;
* все индексные выражения переменных, границы изменения параметров циклов и условия передачи управления задаются, в общем случае, неоднородными формами, линейными как по параметрам циклов, так и по внешним переменным программы; все коэффициенты линейных форм являются целыми числами;
* внешние переменные программы всегда целочисленные, и вектора их значений принадлежат некоторым целочисленным многогранникам; конкретные значения внешних переменных известны только перед началом работы программы и неизвестны в момент ее исследования.

44. Теорема о построении графа алгоритма для линейного класса программ.

Теорема (о возможности построения графа алгоритма)

Если фрагмент программы принадлежит линейному классу, то для него возможно построение графа алгоритма: $(N, \Delta(N), F(N,\Delta))_k$, где

N – линейный многогранник в пространстве внешних переменных (непустой)

$\Delta(N) \neq 0$ – линейный многогранник в пространстве итераций

$F(N,\Delta)$ – векторная функция, описывающая дуги для вершин из $\Delta(N)$; она описывает координаты поставщика информации для текущей операции.

45. Эквивалентные преобразования программ. Преобразования циклов (перестановка, распределение).

У эквивалентных программ должны быть изоморфные графы алгоритмов.

Перестановка циклов.

```
for (i=0; i<n; i++)
        for (j=0;j<n;j++)
                a[i,j]=a[i,j]+a[i,j-1]
```

```
for (j=0; j<n; j++)
        for (i=0;i<n;i++)
                a[i,j]=a[i,j]+a[i,j-1]
```

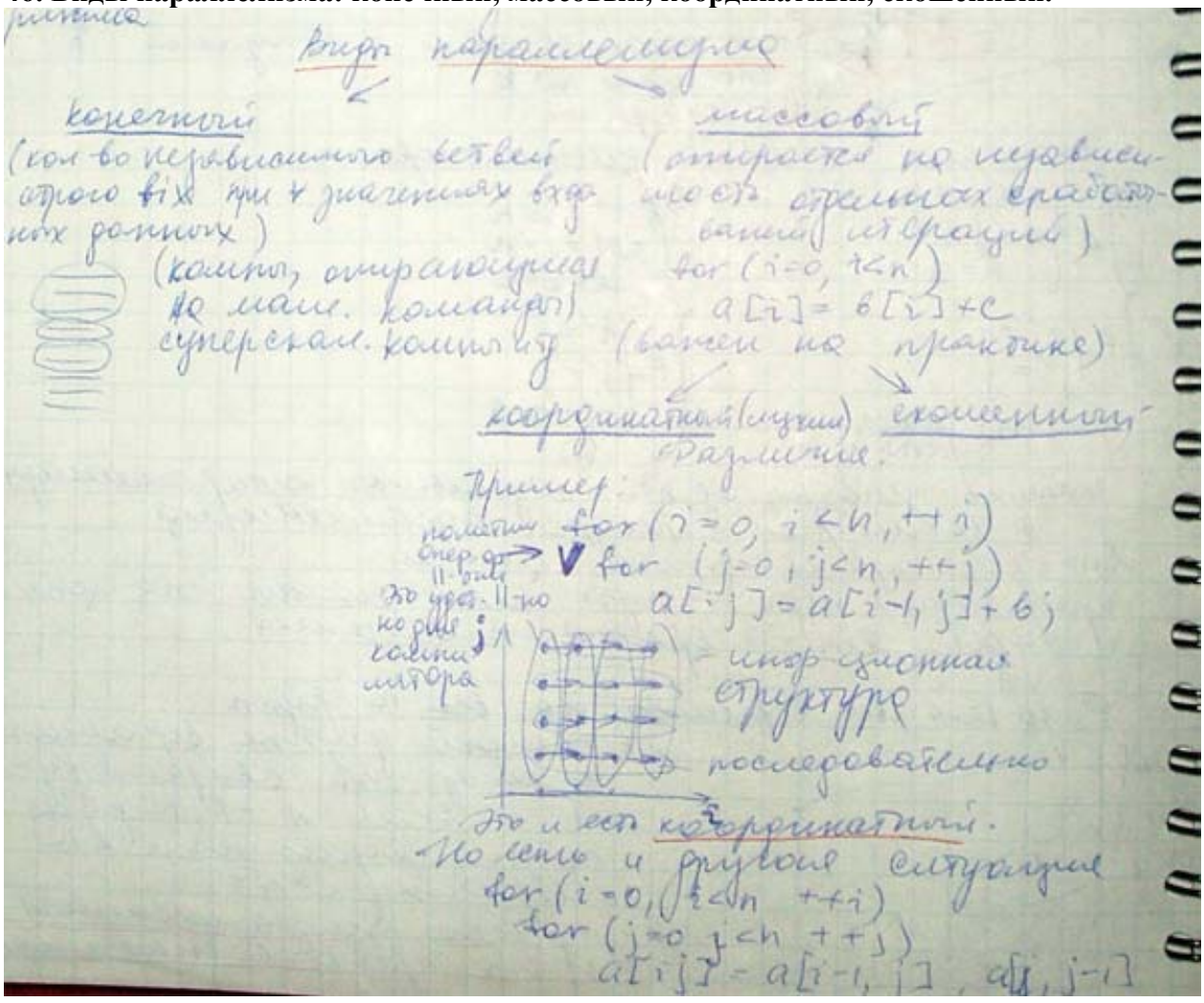
Данные циклы эквивалентны с точки зрения конечного результата

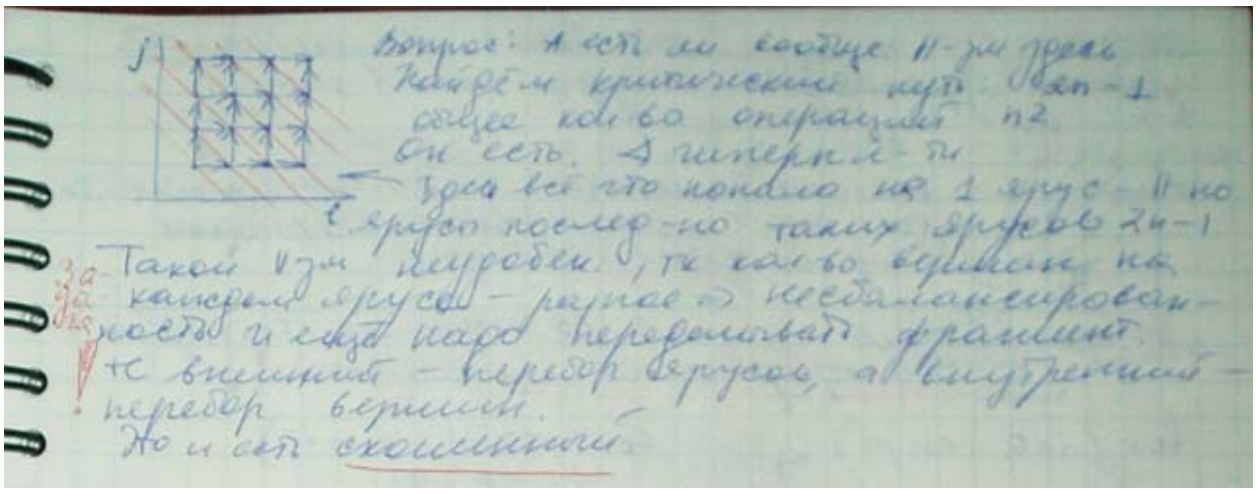
Распределение циклов.

```
for (i=0,i<n;++i) {
  a[i]=a[i-1]+b;
  c[i]=a[i]+c[i]+e;
}
```

```
for (i=0,i<n;++i)
  a[i]=a[i-1]+b;
for (i=0,i<n;++i)
  c[i]=a[i]+c[i]+e;
```

46. Виды параллелизма: конечный, массовый, координатный, скошенный.





47. Ярусно-параллельная форма графа алгоритма, высота, ширина. Каноническая ЯПФ.

Итак, каждое описание алгоритма порождает ориентированный ациклический мультиграф. Верно и обратное. Если задан ориентированный ациклический мультиграф, то его всегда можно рассматривать как граф некоторого алгоритма. Для этого каждой вершине нужно поставить в соответствие любую однозначную операцию, имеющую столько аргументов, сколько дуг входит в вершину. Поэтому между алгоритмами и рассматриваемыми графами есть определенное взаимное соответствие.

Утверждение 4.1

Пусть задан ориентированный ациклический граф, имеющий n вершин. Существует число s < n, для которого все вершины графа можно так пометить одним из индексов 1, 2, ..., s, что если дуга из вершины с индексом i идет в вершину с индексом j, то i<j.

Выберем в графе любое число вершин, не имеющих предшествующих, и пометим их индексом 1. Удалим из графа помеченные вершины и инцидентные им дуги. Оставшийся граф также является ациклическим. Выберем в нем любое число вершин, не имеющих предшествующих, и пометим их индексом 2. Продолжая этот процесс, в конце концов, исчерпаем весь граф. Так как при каждом шаге помечается не менее одной вершины, то число различных индексов не превышает числа вершин графа.

Отсюда следует, что никакие две вершины с одним и тем же индексом не связаны дугой. Минимальное число индексов, которым можно пометить все вершины графа, на 1 больше длины его критического пути. И, наконец, для любого целого числа s, не превосходящего общего числа вершин, но большего длины критического пути, существует такая разметка вершин графа, при которой используются все s индексов.

Граф, размеченный в соответствии с утверждением 4.1, называется **строгой параллельной формой графа**. Если в параллельной форме некоторая вершина помечена индексом k, то это означает, что длины всех путей, оканчивающихся в данной вершине, меньше к. Существует строгая параллельная форма, при которой максимальная из длин путей, оканчивающихся в вершине с индексом k, равна k-1. Для этой параллельной формы число используемых индексов на 1 больше длины критического пути графа. Среди подобных параллельных форм существует такая, в которой все входные вершины находятся в группе с одним индексом, равным 1. Эта строгая параллельная форма называется **канонической**. Для заданного графа его каноническая параллельная форма единственна. Группа вершин, имеющих одинаковые индексы, называется **ярусом** параллельной формы, а число вершин в группе — шириной яруса. Число ярусов в параллельной форме называется **высотой** параллельной формы, а максимальная ширина ярусов — ее **шириной**. Параллельная форма минимальной высоты называется максимальной.

48. Зависимость степени параллелизма от формы записи алгоритма (на примере реализации метода Гаусса).