1 Виды параллельной обработки данных, их особенности

Существует два способа параллельной обработки данных: **параллелизм** и **конвейерность.**

Параллельная обработка:

Предположим, что нам нужно найти сумму двух векторов, состоящих из 100 вещественных чисел каждый. В нашем распоряжении есть устройство, которое выполняет суммирование пары чисел за пять тактов работы компьютера. Устройство сделано так, что на все время выполнения данной операции оно блокируется, и никакой другой полезной работы выполнять не может. В таких условиях вся операция будет выполнена за 500 тактов.

Теперь допустим, что у нас есть два точно таких же устройства, которые могут работать одновременно и независимо друг от друга. Для простоты будем рассматривать идеальную ситуацию, когда нет никаких дополнительных накладных расходов, связанных с получением устройствами входных данных и сохранением результатов. Постоянно нагружая каждое устройство элементами входных массивов, можно получить искомую сумму уже за 250 тактов — ускорение выполнения работы в два раза. В случае 10 подобных устройств время получения результата составит всего 50 тактов, а в общем случае система из N устройств затратит на суммирование время около 500/N

Очень много примеров параллельной обработки можно найти и в нашей повседневной жизни: множество кассовых аппаратов в супермаркете, многополосное движение но оживленным автотрассам, бригада землекопов, несколько бензиновых колонок на автозаправочных станциях и т. п. Кстати, одним из пионеров в параллельной обработке был академик А. А. Самарский, выполнявший в начале 50-х годов прошлого столетия расчеты, необходимые для моделирования ядерных взрывов. Александр Андреевич решил эту сдачу, посадив несколько десятков барышень с арифмометрами за столы. Барышни передавали данные друг другу просто на стоках и откладывали необходимые цифры на арифмометрах. С помощью вот такой «параллельной вычислительной системы», в частности, была рассчитана эволюция взрывной волны.

Конвейерная обработка:

Сложение каждой пары чисел выполняется в виде последовательности микроопераций, таких как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.п. И что примечательно: в процессе обработки каждой пары входных данных микрооперации задействуются только один раз и всегда в одном и том же порядке: одна за другой. Это, в частности, означает, что если первая микрооперация выполнила свою работу и передала результаты второй, то для обработки текущей пары она больше не понадобится, и значит, она вполне может начинать обработку следующей ждущей на входе устройства пары аргументов.

Исходя их этих рассуждений, сконструируем устройство следующим образом. Каждую микрооперацию выделим в отдельную часть устройства и расположим их в порядке выполнения. В первый момент времени входные данные поступают для обработки первой частью. После выполнения первой микрооперации первая часть передает результаты свой работы второй части, а сама берет новую пару. Когда входные аргументы пройдут через все этапы обработки, на выходе устройства появится результат выполнения операции.

Такой способ организации вычислений и носит название конвейерной обработки. Каждая часть устройства называется *ступенью конвейера*, а общее число ступеней — *длиной конвейера.* Предположим, что для выполнения операции сложения вещественных чисел спроектировано конвейерное устройство, состоящее из пяти ступеней, срабатывающих за один такт каждая. Время выполнения одной операции конвейерным устройством равно сумме времен срабатывания всех ступеней конвейера. Это значит, что одна операция сложения двух чисел выполнится за пять тактов, т.е. за столько же времени, за сколько такая же операция выполнялась и на последовательном устройстве в предыдущем случае.

Теперь рассмотрим процесс сложения двух массивов. Как и прежде, через пять тактов будет получен результат сложения первой пары. Но заметим, что одновременно с первой парой прошли частичную обработку и другие элементы. Каждый последующий такт (!) на выходе конвейерного устройства будет появляться сумма очередных элементов. На выполнение всей операции потребуется 104 такта, вместо 500 тактов при использовании последовательного устройства — выигрыш во времени почти в пять раз.

Приблизительно так же будет и в общем случае. Если конвейерное устройство содержит ***l*** ступеней, а каждая ступень срабатывает за одну единицу времени, то время обработки *n* независимых операций этим устройством составит *l* + *n* — 1 единиц. Если это же устройство использовать в монопольном режиме (как последовательное), то время обработки будет равно *l*n.* В результате для больших значений *n* получили ускорение почти в *l* раз за счет использования конвейерной обработки данных.

2 История появления параллелизма в архитектуре ЭВМ: IBM 701, 704, 708, IBM STRETCH, ATLAS

IBM 701 (1953), IBM 704 (1955): разрядно-параллельная память, разрядно- параллельная арифметика.
Первые компьютеры (EDSAC, EDVAC, UNIVAC, конец 40-х — начало 50-х годов XX столетия) для реализации принципа хранимой программы использовали ртутные линии задержки. При таком способе организации памяти разряды слова поступали для последующей обработки последовательно один за другим. Вполне естественно, что точно так же поразрядно выполнялись и арифметические операции, поэтому для сложения двух 32-разрядных чисел требовалось около 32 машинных тактов. Время *разрядно-последовательной* памяти и *разрядно-последовательной* арифметики в истории вычислительной техники.

С изобретением запоминающих устройств с произвольной выборкой данных стала реализуемой как *разрядно-параллельная* память, так и *разрядно-параллельная* арифметика. Все разряды слова одновременно считываются из памяти и участвуют в выполнении операции арифметико-логическим устройством. В 1953 году появилась машина IBM 701 — первая коммерческая ЭВМ, построенная на таком принципе. Однако наиболее удачной машиной этого класса стала выпущенная в 1955 году ЭВМ IBM 704, в которой была впервые применена память на ферритовых сердечниках и аппаратное АУ с плавающей точкой. Удивительный по тем временам коммерческий успех IBM 704 определялся продажей 150 (!) экземпляров этой машины.

IBM 709 (1958): независимые процессоры ввода/вывода.

Как в машине IBM 704, так и в других компьютерах того времени, все операции ввода/вывода осуществлялись через арифметико-логическое устройство. Внешних устройств было немного, но самое быстрое из них — лентопротяжное устройство, работало со скоростью около 15 000 символов в секунду, что было во много раз меньше скорости обработки данных процессором. Подобная организация вычислительных машин вела к существенному снижению производительности во время ввода и вывода информации. Одним из первых решений этой проблемы стало введение специализированной ЭВМ, называемой каналом ввода/вывода, которая позволяла *АЛУ работать параллельно с устройствами ввода/вывода.* В 1958 году к машине IBM 704 добавили шесть каналов ввода/вывода, что явилось основой для построения ЭВМ IBM 709.

IBM STRETCH (1961): опережающий просмотр вперед, расслоение памяти.
Одной из первых машин, воплотивших идею повышения производительности за счет совмещения во времени различных этапов выполнения соседних команд, была система IBM STRETCH. Первый из семи построенных компьютеров этой серии был установлен в Лос-Аламосской национальной лаборатории США в 1961 году. Архитектура системы имела две принципиально важных особенности. Первая особенность — это *опережающий просмотр* для считывания, декодирования, вычисления адресов, а также предварительная выборка операндов нескольких команд.

Вторая особенность заключалась в разбиении памяти на два независимых банка, способных передавать данные в АЛУ независимо друг от друга (то есть опять-таки идеи параллелизма). Эта особенность, получившая название *расслоение памяти*, была позднее использована почти во всех больших вычислительных системах. Основная задача, решаемая в данном случае разработчиками компьютеров, состоит в согласовании скорости работы АЛУ и памяти. Это делается за счет разделения всей памяти на K независимых *банков* (иногда их называют секциями), что позволяет совместить во времени различные обращения к памяти. Одновременно вводится чередование адресов, при котором любые K подряд расположенных ячеек памяти с адресами A_0, $A_0 + 1$, ..., $A_0 + \{K — 1)$ попадают в разные банки. При такой организации работа с подряд расположенными данными проходит максимально эффективно.

ATLAS (1963): конвейер команд.
Одновременно с проектом IBM STRETCH в 1956 году начался и другой известный проект, завершившийся запуском в 1962 году первого компьютера ATLAS. Наверняка, многие слышали о данном проекте, поскольку ATLAS по праву считается значительной вехой в истории развития вычислительной техники. Достаточно сказать, что в нем впервые была реализована мультипрограммная операционная система, основанная на виртуальной памяти и системе прерываний. Для нас же этот компьютер интересен, прежде всего, тем, что в нем впервые начали использовать *конвейерный принцип обработки команд.* Цикл обработки команды разбили на четыре ступени: выборка команды, вычисление адреса операнда, выборка операнда и выполнение операции. Это позволяло, в благоприятной ситуации, сократить среднее время выполнения команды с 6 до 1,6 мкс и увеличить производительность компьютера до 200 тысяч вещественных операций в секунду.

3 История появления параллелизма в архитектуре ЭВМ: CDC 6600, CDC 7600, ILLIAC IV

CDC 6600: независимые функциональные устройства

В 1964 году компания Control Data Corporation выпускает компьютер CDC 6600. Генеральным конструктором компьютера, а на то время и вице-президентом компании, был С. Крэй, создавший впоследствии множество уникальных суперкомпьютеров. Центральный процессор CDC 6600 содержал *десять независимых функциональных устройств* — параллелизм в явном виде. Все функциональные устройства могли работать одновременно друг с другом и являлись специализированными (устройства сложения с плавающей и фиксированной запятой (по одному), умножения (два устройства), деления, логических операций, сдвига и т. п.). Чтобы лучше почувствовать возможности суперкомпьютера того времени, приведем некоторые характеристики CDC 6600: время такта 100 нс, память разбита на 32 банка по 4096 60разрядных слов, средняя производительность 2—3 млн. операций в секунду.

CDC 7600 (1969): конвейерные независимые функциональные устройства

В 1969 году Control Data Corporation выпускает компьютер CDC-7600 — усовершенствованный вариант модели CDC-6600. Центральный процессор нового компьютера содержит *восемь независимых конвейерных функциональных устройств* — одновременное использование и параллелизма, и конвейерности. В отличие от предшественника, в CDC-7600 реализована двухуровневая память: вычислительная секция в обычном режиме работает со "сверхоперативной" памятью (64К 60-разрядных слов), куда при необходимости подкачиваются данные из основной памяти (512К 60-разряцных слов). Время такта CDC-7600 равно 27,5 не, средняя производительность 10—15 млн. операций в секунду.

ILLIAC IV (1974): матричные процессоры

Однако чудеса в жизни редко случаются... Проект компьютера ILLIAC IV слишком опередил свое время, что и предопределило постоянные задержки и корректировки планов. Работа над системой началась в 1967 году. В 1972 году изготовили один квадрант из 64 процессорных элементов, который был установлен в научно-исследовательском центре NASA Ames (США). Наладка системы длилась до 1974 года, после чего ее сдали в эксплуатацию, но работы по доводке продолжались до 1975 года. В реальном использовании система находилась до 1982 года, после чего этот единственный изготовленный экземпляр нашел свое последнее пристанище в музее.

Что же получилось в результате? Планировали матрицу из четырех квадрантов и 256 процессорных элементов, а сделали только один, проектное время такта по технологическим соображениям пришлось увеличить до 80 нс, стоимость изготовления четвертой части системы оказалась в четыре раза выше, чем предусматривалось сначала, а ее реальная производительность достигала лишь 50 млн. вещественных операций в секунду. Полное фиаско? Ни в коей мере. Данный проект оказал огромное влияние как на архитектуру последующих поколений компьютеров, так и на развитие многих компонентов программного обеспечения параллельных вычислительных систем. Отработанные в рамках проекта технологии были использованы при создании систем PEPE, BSP, ICL DAP и многих других.

4 Оценка вычислительной сложности

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 $5{,}3*10^6$ ($5{,}256{,}000 = 5{,}2*10^6$) 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 106 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.

5 Микроэлектроника и архитектура: оценка вклада в увеличение производительности компьютеров.

Бесспорно, успехи микроэлектроники впечатляют. Согласно *закону Мура*, не имеющего строгого доказательства, но подтверждающегося уже не один десяток лет, производительность "обычных" компьютеров удваивается каждые полтора года. Однако попытаемся разобраться, только ли этими причинами определяются те рекордные показатели производительности, которыми обладают современные компьютеры. Обратимся к известным историческим фактам и проведем простое сравнение (таблица). На одном из первых компьютеров мира — EDSAC, появившемся в 1949 году и имевшем время такта 2 микросекунды, можно было выполнить 2*n* арифметических операций за 18й миллисекунд, т. е. в среднем 100 арифметических операций в секунду. Сравним эти данные, например, с характеристиками одного вычислительного узла современного суперкомпьютера HP Superdome время такта приблизительно 1,3 нс (процессор PA-8700, 750 МГц), а пиковая производительность около 192 миллиардов арифметических операций в секунду.

Что же получается? За полвека производительность компьютеров выросла почти в два миллиарда раз. При этом выигрыш в быстродействии, связанный с уменьшением времени такта с 2 мкс до 1,3 нс, составляет лишь около 1500 раз. С этой цифрой естественно связать развитие микроэлектроники (что верно, конечно же, лишь в некотором приближении). Откуда же взялось остальное? Ответ достаточно очевиден — использование новых решений в архитектуре компьютеров, и среди них далеко не последнее место занимает принцип параллельной обработки данных, воплошаюший идею одновременного выполнения нескольких действий.

Удивительно, но несмотря на все многообразие форм проявления параллелизма в архитектуре компьютеров, на обилие сопутствуюших проблем, сушествует лишь два способа параллельной обработки данных, собственно параллелизм и конвейерность.

Таблица	EDSAC 1949 год	***HP Superdome****2001 год*
тактовая частота	0,5 МГц	770 МГц
производительность	10^2оп/с	$1{,}9 \times 10^9$оп/с

6 Архитектура и параметры суперкомпьютерных систем – лидеров списка Top500 (примеры)

TOP500 — проект по составлению рейтинга и описаний 500 самых мощных общественно известных вычислительных систем мира. Проект был запущен в 1993 году и публикует актуальный перечень суперкомпьютеров дважды в год (в июне и ноябре). Этот проект направлен на обеспечение надёжной основы для выявления и отслеживания тенденций в области высокопроизводительных вычислений. Основой для рейтинга являются результаты исполнения испытания LINPACK (HPL), решающего большие СЛАУ.

Положение	**Rmax** **Rpeak** **(Pflops)**	**Название**	**Архитектура** **Тип процессора, сеть**	**Операционная система**
1	33,863 54,902	*Tianhe-2*	**NUDT** Xeon E5-2692, Xeon Phi, Custom	Linux (Kylin)
2	17,590 27,113	*Titan*	**Cray XK7** Opteron 6274, Tesla K20X, Custom	Linux (CLE, SLES based)
3	17,173 20,133	*Sequoia*	**Blue Gene/Q** PowerPC A2, Custom	Linux (RHEL, CNK)
4	10,510 11,280	*K computer*	**RIKEN** SPARC64 VIIIfx, Tofu	Linux
5	8,586 10,066	*Mira*	**Blue Gene/Q** PowerPC A2, Custom	Linux (RHEL, CNK)
6	6,271 7,788	*Piz Daint*	**Cray XC30** Xeon E5-2670, Tesla K20X, Aries interconnect	Linux (CLE, SLES based)
7	5.537 7.235	*Shaheen II*	**Cray XC40** Xeon E5-2698v3, Aries	Linux (CLE)

8	5,168 8,520	*Stampede*	**PowerEdge C8220** Xeon E5-2680, Infiniband	Linux
9	5,008 5,872	*JUQUEEN*	**Blue Gene/Q** PowerPC A2, Custom	Linux (RHEL, CNK)
10	4,293 5,033	*Vulcan*	**Blue Gene/Q** PowerPC A2, Custom	Linux (RHEL, CNK)

Суперкомпьютеры и их характеристики (2015 г.)

Tianhe-2, 16 000 вычислительных узлов:
2 * Intel XeonE5-2692 (12cores), 2.93GHz + 3 *Xeon Phi, 33.86 Pflop/s, ОП = 1,375 PB + 375 TB, HDD = 12 PB
Всего: 3 120 000 ядер
Производительность: Peak: 54,9 Pflop/s Linpack: 33,86 Pflop/s

Cray XK7, Titan, 18 688 вычислительных узлов:
AMD Opteron 16-core + NVIDIA Tesla K20 17.59 Pflop/s, ОП = 710 TB, HDD = 10 PB
299 008 ядер
Производительность: 17,59 Pflop/s

IBM BlueGene/Q, Sequoia, 1 572 864 ядра, IBM PowerPC A2, 1.6 GHz 16.32 Pflop/s, ОП = 1.57 PB, HDD = 55 PB, Tape = “virtually unlimited”
1 572 864 ядра
Производительность: 16.32 Pflop/s

IBM RoadRunner, 6562 AMD Opteron DC + 12240 IBM Cell, 1.042 Pflop/s, ОП = 98 TB
122 400 процессоров
Производительность: 1,042 Pflop/s.

Т-Платформы, Ломоносов, 52 168 x86-cores, 2 130 NVIDIA X2070 Intel Xeon 5570 (4cores), 5670 (6cores) 2.93 GHz 901 Tflop/s, ОП = 92 TB, HDD = 650 TB, Tape = 1.2 PB

7 Список Top500: принципы формирования, структура, параметры

TOP500 — проект по составлению рейтинга и описаний 500 самых мощных общественно известных вычислительных систем мира. Проект был запущен в 1993 году и публикует актуальный перечень суперкомпьютеров дважды в год (в июне и ноябре). Этот проект направлен на обеспечение надёжной основы для выявления и отслеживания тенденций в области высокопроизводительных вычислений. Основой для рейтинга являются результаты исполнения испытания LINPACK (HPL), решающего большие СЛАУ.

Что имело бы смысл взять в качестве теста для вычисления эффективности работы вычислительной системы? Что-то несложное и известное всем. Таким тестом стал Linpack. Эта программа предназначена для решения системы линейных алгебраических уравнений с плотной матрицей с выбором главного элемента по строке. Простой алгоритм, регулярные структуры данных, значительная вычислительная емкость, возможность получения показателей производительности, близких к пиковым, — все эти черты сделали тест исключительно популярным.

Этот тест рассматривается в трех вариантах. В первом варианте решается система с матрицей размера 100x100. Предоставляется уже готовый исходный текст, в который не разрешается вносить никаких изменений. Изначально предполагалось, что запрет внесения изменений в программу не позволит использовать каких-либо специфических особенностей аппаратуры, и показатели производительности будут сильно занижены. В настоящее время ситуация полностью изменилась. Матрица столь небольшого размера легко помещается целиком в кэш-память современного процессора (нужно лишь 80 Кбайт), что завышает его характеристики.

Во втором варианте теста размер матрицы увеличивается до 1000x1000. Разрешается как внесение изменений в текст подпрограммы, реализующей заложенный авторами метод решения системы, так и изменение самого метода. Единственное ограничение — это использование стандартной вызывающей части теста, которая выполняет инициализацию матрицы, вызывает подпрограмму решения системы и проверяет корректность результатов.

В этой же части вычисляется и показанная компьютером производительность, исходя из формулы для числа операций $2n^3/3 + 2n^2$ (*n* — это размер матрицы), вне зависимости от вычислительной сложности реально использованного алгоритма.

В данном варианте достигались значения производительности, близкие к пиковой. Этому способствовали и значительный размер матрицы, и возможность внесения изменений в программу или алгоритм. Отсюда появилось и специальное название данного варианта — Linpack TPP, Toward peak perfomance.

С появлением больших массивно-параллельных компьютеров вопрос подбора размера матрицы стал исключительно актуальным. На матрицах 100x100 или 1000x1000 никаких разумных показателей получить не удавалось. Эти задачи были слишком малы. Матрица размера 1000x1000 занимает лишь 0,01—0,001% всей доступной оперативной памяти компьютера. Первые эксперименты с тестом Linpack на реальных массивно-параллельных компьютерах показали, что и фиксировать какой-либо один размер задачи тоже нельзя. В результате в третьем варианте теста было разрешено использовать изложенную во втором варианте методику для матриц сколь угодно большого размера. Сколько есть оперативной памяти на всех вычислительных узлах системы, столько и можно использовать. Для современных компьютеров размер матрицы уже перевалил за миллион, поэтому такой шаг был просто необходим.

Для работы теста Linpack на вычислительных системах с распределенной памятью была создана специальная версия HPL – High Perfomance Linpack. В отличие от стандартной реализации, в HP1_ пользователь может управлять большим числом параметров для достижения высокой производительности на своей установке.

В настоящее время Linpack используется для формирования списка Top500 — пятисот самых мощных компьютеров мира **(**www.top500.org**).** Кроме пиковой производительности $R_{реак}$ для каждой системы указывается величина R_{max}, равная производительности компьютера на тесте Linpack с матрицей максимального для данного компьютера размера R_{max}. По значению R_{max} отсортирован весь список. Как показывает анализ представленных данных, величина R_{max} составляет 50—70% от значения $R_{реак}$. Интерес для анализа функционирования компьютера представляет и указанное в каждой строке значение $N_{1/2}$, показывающее, на матрице какого размера достигается половина производительности R_{max}.

Что же показывают данные теста Linpack*? Только одно — насколько хорошо компьютер может решать системы уравнений с плотной матрицей указанным методом.* Поскольку задача имеет хорошие свойства, то и корреляция производительности на тесте Linpack с пиковой производительностью компьютера высока. Операции ввода/вывода не затрагиваются, отношение числа выполненных операций к объему используемых данных высокое, регулярность структур данных и вычислений, простая коммуникационная схема, относительно небольшой объем передач между процессорами и другие свойства программы делают данный тест "удобным". Безусловно, по данным этого теста можно получить много полезной информации о вычислительной системе и с этим никто не спорит. Но нужно ясно понимать и то,

что высокая производительность на ЫШРАСК совершенно не означает того, что и на вашей конкретной программе будет достигнута высокая производительность.

Интересное свойство Linpack связано с законом Мура. Согласно этому закону производительность вычислительных систем удваивается каждые 18 месяцев. Не имея строгих доказательств, этот закон, тем не менее, подтверждается уже не один десяток лет. Согласно закону, объем памяти увеличивается в четыре раза каждые три года. Это позволит каждые три года удваивать размер используемой матрицы. За эти же три года производительность компьютеров вырастет в четыре раза. Поскольку вычислительная сложность теста Linpack есть куб от размера матрицы, то время выполнения третьего варианта теста должно удваиваться каждые три года. Получается, что фиксировать размер матрицы нельзя, а использование матриц максимального размера со временем приведет к очень большим затратам. Желателен компромисс, который пока не найден.

8 Иерархия памяти, локальность вычислений, локальность использования данных

Анализируя архитектурные особенности компьютеров, мы неоднократно, хотя, быть может, и неявно, затрагивали очень важную проблему согласования скорости работы процессора и времени выборки данных из памяти. В одних компьютерах использовалось расслоение памяти, в других вводилась развитая регистровая структура, многоуровневая память, кэш-память или что-то еще, но в любом случае все подобные особенности организации памяти направлены на достижение одной цели — ускорить выборку команд и данных.

Давно было подмечено, что процесс выполнения большинства программ характеризуется двумя свойствами: *локальностью вычислений* и *локальностью использования данных.* И в том, и в другом случае следующий необходимый для работы программы объект (команда или операнд) расположен в оперативной памяти "недалеко" от предыдущего. Идеальный пример фрагмента программы с высокой локальностью вычислений — это цикл с телом из одного оператора и большим числом итераций. На практике многие программы обладают сразу двумя свойствами, и локальностью вычислений, и локальностью использования данных. Но, к сожалению, это ни в коей мере не является законом. Вызовы подпрограмм и функций, косвенная адресация массивов, неудачная работа с многомерными массивами и сложными структурами данных, использование разного рода условных операторов — это лишь небольшой список причин, по которым свойства локальности в реальных программах могут значительно ослабевать.

Основной конструкцией в языках программирования, определяющей как локальность вычислений, так и локальность использования данных, являются циклы. В идеале, все тело цикла лучше поместить в более быструю память, поскольку на следующей итерации, скорее всего, нужно будет выполнить те же самые команды. Похожая ситуация и с данными: все то, что используется часто, лучше хранить на регистрах, у которых времена чтения/записи всегда согласованы со скоростью работы процессора. Однако удовольствие это очень дорогое и сколько-нибудь значительные объемы данных на регистрах держать невозможно. В такой ситуации будет использоваться следующий уровень в *иерархии памяти*, например, кэш-память. Если объема кэш памяти не хватит, то можно задействовать следующий уровень и т. д. Аналогий этому факту очень много в повседневной жизни. Все самое необходимое мастер всегда держит под рукой на рабочем столе (регистры), часто используемые предметы хранятся здесь же в ящиках стола (кэш-память), основной набор инструментов лежит в шкафу недалеко от стола (основная память), за чем-то приходится иногда спускаться на склад (дисковая память), а что-то время от времени приходится заказывать даже из другой мастерской (архив на магнитной ленте).

Примеров конкретных реализаций различных уровней памяти можно привести очень много. Это регистры и регистровые файлы, кэш-память разных уровней, основная оперативная память, виртуальные диски, реализованные либо на отдельных участках основной оперативной памяти, либо на другой, более медленной памяти, жесткие диски, ленточные роботы и многое многое другое. А закон формирования иерархии памяти в каждом компьютере один: чем выше уровень в иерархии, тем выше скорость работы с данными, тем дороже (в пересчете на слово или байт) обходится память, поэтому объем каждого последующего уровня становится все меньше и меньше.

Эффективность использования иерархии памяти подметили давно. Так, даже в описанном выше компьютере ILLIAC IV уже можно выделить, по крайней мере, четыре уровня. Каждый процессорный элемент имел по 6 программно-адресуемых регистров и собственную локальную оперативную память на 2048 слов. Для хранения данных использовались два жестких диска на 1 Гбит каждый, а для долговременного хранения предназначалась лазерная память с однократной записью емкостью в 10^{12} бит (луч аргонового лазера выжигал отверстия в тонкой металлической пленке, укрепленной на барабане).

Безусловно, иерархия памяти не имеет прямого отношения к параллелизму. Но она относится к тем особенностям архитектуры компьютеров, которые имеют огромное значение для повышения их производительности, поэтому ничего не сказать о ней мы, конечно же, не могли. Это тем более справедливо, если учесть, что в настоящее время подобная иерархия поддерживается везде, от суперкомпьютеров до персональных компьютеров.

9 Закон Амдала, его следствия, суперлинейное ускорение

Назовем функциональное устройство простым, если никакая последующая операция не может начать выполняться раньше, чем закончится предыдущая.

Назовем стоимостью операции время ее реализации, а стоимостью работы — сумму стоимостей всех выполненных операций. Стоимость работы — это время последовательной реализации всех рассматриваемых операций на простых ФУ с аналогичными временами срабатываний. Загруженностью устройства на данном отрезке времени будем называть отношение стоимости реально выполненной работы к максимально возможной стоимости. Ясно, что загруженность p всегда удовлетворяет условиям $0 < p < 1$ Имеет место также очевидное

Утверждение 2.1

Максимальная стоимость работы, которую можно выполнить за время T равна T для простого ФУ и nT для конвейерного ФУ длины n

Будем называть реальной производительностью системы устройств количество операций, реально выполненных в среднем за единицу времени. Пиковой производительностью будем называть максимальное количество операций, которое может быть выполнено той же системой за единицу времени при отсутствии связей между ФУ. Из определений вытекает, что как реальная, так и пиковая производительности системы суть суммы соответственно реальных и пиковых производительностей всех составляющих систему устройств.

.

Утверждение 2.2

Пусть система состоит из s устройств, в общем случае простых или конвейерных. Если устройства имеют пиковые производительности $\pi_1, \ldots, \pi_s$ и работают с загруженностями $p_1, \ldots, p_s$ то реальная производительность r системы выражается формулой

$r = \sum_{i=1}^{s} p_i \pi_i$

r, π, p - соответственно реальная производительность, пиковая производительность и загруженность одного устройства.

Утверждение 2.3

Если система состоит из s устройств одинаковой пиковой производительности простых или конвейерных, то:

- загруженность системы равна среднему арифметическому загруженностей всех устройств;
- реальная производительность системы равна сумме реальных производительностей всех устройств;
- пиковая производительность системы в s раз больше пиковой производительности одного устройства;
- ускорение системы равно сумме загруженностей всех устройств;
- если система состоит только из простых устройств, то ее ускорение равно отношению времени реализации алгоритма на одном универсальном простом устройстве с той же пиковой производительностью к времени реализации алгоритма на системе.

Снова рассмотрим систему из s устройств. Не ограничивая общности, будем считать все устройства простыми, т. к. любое конвейерное ФУ всегда можно представить как линейную цепочку простых устройств. Допустим, что между устройствами установлены направленные связи, и они не меняются в процессе функционирования системы. Построим ориентированный мультиграф, в котором вершины символизируют устройства, а дуги — связи между ними. Дугу из одной вершины будем проводить в другую в том и только том случае, когда результат каждого срабатывания устройства, соответствующего первой вершине, обязательно передается в качестве аргумента для очередного срабатывания устройству, соответствующему второй вершине. Назовем этот мультиграф графом системы. Предположим, что каким-то образом в систему введены все исходные данные и она начала функционировать согласно описанным ранее правилам. Если в процессе функционирования какие-то ФУ будут требовать для своих срабатываний другие исходные данные, то будем предполагать, что они подаются на входы ФУ без задержек. Исследуем максимальную производительность системы, т. е. ее максимально возможную реальную производительность при достаточно большом времени функционирования.

Утверждение 2.4

Пусть система состоит из s простых устройств с пиковыми производительностями π_1, …, π_s. Если граф системы связный, то максимальная производительность r_{max} системы выражается формулой

$$r_{max} = s \min_{1 \le i \le s} \pi_i$$

Следствие

Пусть вычислительная система состоит из s простых устройств с пиковыми производительностями π_1, …, π_s. Если граф системы связный, то:

□ асимптотически каждое из устройств выполняет одно и то же число операций;

□ загруженность любого устройства не превосходит загруженности самого непроизводительного устройства;

□ если загруженность какого-то устройства равна 1, то это - самое непроизводительное устройство;

□ загруженность системы не превосходит

$$p_{max} = s \frac{\min_{1 \le i \le s} \pi_i}{\sum_{i=1}^{s} \pi_i}$$

□ ускорение системы не превосходит

$$R_{max} = s \frac{\min_{1 \le i \le s} \pi_i}{\max_{1 \le i \le s} \pi_i}$$

Следствие(1 -ый закон Амдала)

Производительность вычислительной системы, состоящей из связанных между собой устройств, в общем случае определяется самым непроизводительным ее устройством.

Следствие

Пусть система состоит из простых устройств и граф системы связный. Асимптотическая производительность системы будет максимальной, если все устройства имеют одинаковые пиковые производительности.

Допустим, что высота параллельной формы равна m, ширина равна q и всего в алгоритме выполняется N операций.

Утверждение 2.5

В сформулированных условиях максимально возможное ускорение системы равно N/m.

Следствие

Минимальное число устройств системы, при котором может быть достигнуто максимально возможное ускорение, равно ширине алгоритма.

Следствие (2-й закон Амдала)

Пусть система состоит из s одинаковых простых универсальных устройств. Предположим, что при выполнении параллельной части алгоритма все s устройств загружены полностью. Тогда максимально возможное ускорение равно

$$R = \frac{s}{\beta s + (1 - \beta)}$$

Следствие (3-й закон Амдала)

Пусть система состоит из простых одинаковых универсальных устройств. При любом режиме работы ее ускорение не может превзойти обратной величины доли последовательных вычислений.

9 Закон Амдала, его следствие, суперлинейное ускорение

К сожалению, чудеса в жизни редко случаются. Гигантская производительность параллельных компьютеров и супер-ЭВМ с лихвой компенсируется сложностями их использования. Начнем с самых простых вещей. У вас есть программа и доступ, скажем, к 256-процессорному компьютеру. Что вы ожидаете? Да ясно что: вы вполне законно ожидаете, что программа будет выполняться в 256 раз быстрее, чем на одном процессоре. А вот как раз этого, скорее всего, и не будет.

Закон Амдала и его следствия

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 основательно переделывать код для перехода на параллельный компьютер (а любой суперкомпьютер, в частности, является таковым) надо основательно подумать. Если оценив заложенный в программе алгоритм вы поняли, что доля последовательных операций велика, то на значительное ускорение рассчитывать явно 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 тоже 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 одной. 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.

Утверждение 2.2
Пусть система состоит из s устройств, в общем случае простых или конвейерных. Если устройства имеют пиковые производительности π_1, …, π_s и работают с загруженностями p_1, …, p_s то реальная производительность r системы выражается формулой

$r = \sum_{i=1}^{s} p_i \pi_i$ **(2.1)**

Пусть система состоит из s устройств, в общем случае простых или конвейерных. Если устройства имеют пиковые производительности π_1, …, π_s и работают с загруженностями p_1, …, p_s, то будем считать по определению, что ***загруженность*** системы есть величина

$$r = \sum_{i=1}^{s} a_i p_i$$

$$a_i = \frac{\pi_i}{\sum_{j=1}^{s} \pi_j}$$

Большое число ФУ, так же как и конвейерные ФУ, используются тогда, когда возникает потребность решить задачу быстрее. Чтобы понять, насколько быстрее это удается сделать, нужно ввести понятие "***ускорение***". Как и в случае загруженности, оно может вводиться различными способами, многообразие которых зависит от того, что с чем и как сравнивается. Нередко ***ускорение*** определяется, например, как отношение времени решения задачи на одном универсальном процессоре к времени решения той же задачи на системе из s таких же процессоров. Очевидно, что в наилучшей ситуации ускорение может достигать s. **Отношение ускорения к s называется эффективностью.** Заметим, что подобное определение ускорения применимо только к системам, состоящим из одинаковых устройств, и не распространяется на смешанные системы. **Понятие же "эффективность" в рассматриваемом случае просто совпадает с понятием загруженности.** При введении понятия "ускорение" мы поступим иначе.

Пусть алгоритм реализуется за время T на вычислительной системе из s устройств, в общем случае простых или конвейерных и имеющих пиковые производительности π_1, …, π_s. Предположим, что $\pi_1 \leq \pi_2 \leq \ldots \leq \pi_s$. При реализации алгоритма система достигает реальной производительности r из (2.1). Будем сравнивать скорость работы системы со скоростью работы гипотетического простого универсального устройства, имеющего такую же пиковую производительность π_s, как самое быстрое ФУ системы, и обладающего возможностью выполнять те же операции, что все ФУ системы.

Итак, будем называть отношение $R = r / \pi_s$ ускорением реализации алгоритма на данной вычислительной системе или просто ускорением. Выбор в качестве гипотетического простого, а не какого-нибудь другого, например, конвейерного ФУ объясняется тем, что одно простое универсальное ФУ может быть полностью загружено на любом алгоритме. Принимая во внимание (2.1), имеем

$$R = \frac{\sum_{i=1}^{s} p_i \pi_i}{\max\limits_{1 \leq i \leq s} \pi_i} \quad (2.5)$$

Анализ определяющего ускорение выражения (2.5) показывает, что ускорение системы, состоящей из s устройств, никогда не превосходит s и может достигать s в том и только в том случае, когда все устройства системы имеют одинаковые пиковые производительности и полностью загружены.

Ускорение реализации алгоритма на вычислительной системе, состоящей из одинаковых процессоров – отношение времени выполнения алгоритма на одном процессоре (на одном ядре процессора) ко времени 4 параллельного выполнения. Ускорение зависит от выбора параллельного алгоритма и от того, насколько этот алгоритм адекватен архитектуре вычислительной системы.

Эффективность реализации алгоритма на вычислительной системе, состоящей из s одинаковых процессоров – отношение ускорения к s.

Масштабируемость (scalability) — способность системы увеличивать свою производительность при добавлении ресурсов (обычно аппаратных).

Система называется масштабируемой, если она способна увеличивать производительность пропорционально дополнительным ресурсам.

11 Сильная масштабируемость, масштабируемость вширь, слабая масштабируемость. Функция изоэффективности

Масштабируемость (scalability) — способность системы увеличивать свою производительность при добавлении ресурсов (обычно аппаратных).
Система называется масштабируемой, если она способна увеличивать производительность пропорционально дополнительным ресурсам.

Масштабируемость параллельной программы определяется относительно конкретного компьютера и показывает, как изменяются динамические характеристики данной программы при использовании бóльших вычислительных ресурсов.
Масштабируемость можно оценить через отношение прироста производительности системы к приросту используемых ей ресурсов.
Чем ближе это отношение к единице, тем масштабируемость лучше.

Масштабируемость компьютера:
• Вертикальная масштабируемость (масштабируемость вглубь, scale up) – возможность замены платформы, в которой функционирует система, на новую, обладающую большей производительностью.
• Горизонтальная масштабируемость (масштабируемость вширь, scale out) – возможность увеличения производительности системы за счет добавления дополнительных программных или аппаратных средств.

Вычислительная сложность задачи W – количество основных вычислительных шагов лучшего последовательного алгоритма, необходимых для решения задачи на одном процессоре. W является некоторой функцией от размера входных данных. Если для простоты предположить, что каждый основной вычислительный шаг выполняется за единицу времени, то получим W = T1.

Масштабируемость параллельных программ:
• Сильная масштабируемость (strong scaling) — зависимость производительности R от количества процессоров p при фиксированной вычислительной сложности задачи (W = const).
• Масштабируемость вширь (wide scaling) – зависимость производительности R от вычислительной сложности задачи W при фиксированном числе процессоров (p = const)
• Слабая масштабируемость (weak scaling) — зависимость производительности R от количества процессоров p при фиксированной вычислительной сложности задачи в пересчёте на один процессор (W/p = const).

Функция изоэффективности (isoefficiency function):
• Определим, в какой степени должна увеличиваться вычислительная сложность задачи W в зависимости от числа процессоров p, чтобы эффективность распараллеливания E оставалась постоянной. Эта характеристика будет показывать масштабируемость конкретной вычислительной системы.
• Чем меньше необходимая степень роста W для поддержания нужного уровня эффективности распараллеливания, тем более масштабируемой является система. Основные показатели эффективности и масштабируемости параллельных программ
• E = 1/(1+T0 /T1) = 1/(1+T0 /W). • W = E/(1-E)*T0 = K*T0 , где K = E/(1-E)
• Получившаяся зависимость размера задачи, необходимой для достижения заданной эффективности распараллеливания, от числа процессоров – функция изоэффективности (isoefficiency function).

12 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.

Чтобы полностью использовать потенциал данной архитектуры, необходимо решить три основные задачи:

- найти в программе ветви вычислений, которые можно исполнять параллельно;
- распределить данные по модулям локальной памяти процессоров;
- согласовать распределение данных с параллелизмом вычислений.

Если не решить первую задачу, то бессмысленно использовать многопроцессорные конфигурации компьютера. Если решена первая, но не решены последние две задачи, то все время работы программы может уйти на обмен данными между процессорами. В таком случае про масштабируемость программы можно забыть.

13. Компьютеры с общей и распредёленной 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.

Эти два класса, компьютеры с общей и распределенной памятью, появились не случайно. Они отражают две основные задачи параллельных вычислений.

- *Первая задача* заключается в **построении вычислительных систем с максимальной производительностью.** Это легко позволяет сделать компьютеры с распределенной памятью.

Уже сегодня существуют установки, объединяющие тысячи вычислительных узлов в рамках единой коммуникационной среды. Да что говорить, даже Интернет можно рассматривать как самый большой параллельный компьютер с распределенной памятью, объединяющий миллионы вычислительных узлов. Но как такие системы эффективно использовать? Как убрать большие накладные расходы, идущие на взаимодействие параллельно работающих процессоров? Как упростить разработку параллельных профамм? Практически единственный способ профаммирования подобных систем — это использование систем обмена сообщениями, например, PVM или MPI, что не всегда просто.

Отсюда возникает *вторая задача* — **поиск методов разработки эффективного программного обеспечения для параллельных вычислительных систем.** Данная задача немного проще решается для компьютеров с общей памятью. Накладные расходы на обмен данными между процессорами через общую память минимальны, а технологии профаммирования таких систем, как правило, проще. Проблема здесь в другом. По технологическим причинам не удается объединить большое число процессоров с единой оперативной памятью, а потому очень большую производительность на таких системах сегодня получить нельзя.

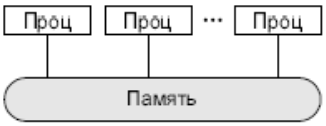


Рис. 2.8. Параллельные компьютеры с общей памятью

Рис. 2.9. Параллельные компьютеры с распределенной памятью

Комп.с общей памятью: С одной стороны, программирование компьютеров этого класса значительно проще, чем, скажем, программирование вычислительных кластеров с распределенной памятью. В самом деле, не нужно думать о распределении массивов, внутренний параллелизм программ описывается просто, процесс отладки идет быстрее. С другой стороны, классические представители этого класса имеют и два недостатка: небольшое число процессоров и очень высокую стоимость.

Чтобы увеличить число процессоров, но сохранить возможность работы в рамках единого адресного пространства, предлагались различные решения (см. § 2.2). Одним из распространенных вариантов является решение на основе архитектуры ccNUMA (cache coherent Non Uniform Memoiy Access). В такой архитектуре память всего компьютера физически распределена, что значительно увеличивает потенциал его масштабируемости. Вместе с тем, память логически остается общей. Это дает возможность использования всех технологий и методов программирования, созданных для SMP-компью-теров. Дополнительно в такой архитектуре содержимое кэш-памяти отдельных процессоров на уровне аппаратуры согласуется с содержимым оперативной памяти (решается проблема когерентности кэшей, cache coherence problem). Значительно увеличивая число процессоров по сравнению с SMP-компьютерами, архитектура ccNUMA привносит дополнительную особенность, несвойственную компьютерам с обшей памятью. Время обрашения к памяти зависит от того, является ли это обрашением к локальной или удаленной памяти. Процесс написания профамм остается прежним, и физическая распределенность памяти профаммисту не видна. Однако ясно, что эффективность профамм напрямую зависит от степени "неоднородности" доступа к памяти.

Комп. с рапред памятью: Идея построения: берется какое-то количество вычислительных узлов, которые объединяются друг с другом некоторой коммуникационной средой. *Вычислительный узел* имеет один или несколько процессоров и свою собственную локальную память, разделяемую этими процессорами. Распределенность памяти означает то, что каждый процессор имеет непосредственный доступ только к локальной памяти своего узла. Доступ к данным, расположенным в памяти других узлов, выполняется дольше и другими, более сложными способами. В последнее время в качестве узлов все чаще и чаще используют полнофункциональные компьютеры, содержащие, например, и собственные внешние устройства. Коммуникационная среда может специально проектироваться для данной вычислительной системы либо быть стандартной сетью, доступной на рынке.

Варианты соединения процессоров с модулями памяти (т.е. в системах с разделяемой памятью): общая шина, матричный коммутатор, каскадные переключатели(омега сеть)

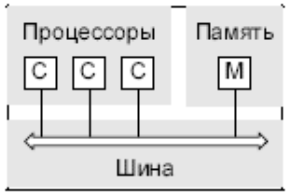


Рис. 2.10. Мультипроцессорная система с общей шиной

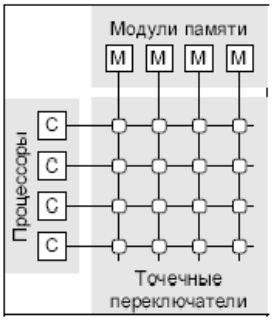


Рис. 2.11. Мультипроцессорная система с матричным коммутатором

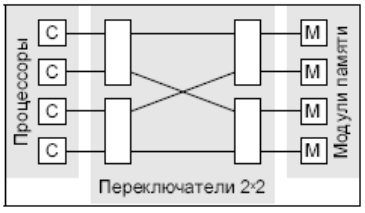


Рис. 2.12. Мультипроцессорная система с омега-сетью
с каскадными переключателями

14. UMA, NUMA и ccNUMA архитектуры. Компьютеры Cm*, BBNButterfly.

NUMA-компьютер (создали разработчики системы Cm*)состоит из набора кластеров, соединенных друг с другом через межкластерную щину. Каждый кластер объединяет процессор, контроллер памяти, модуль памяти и, быть может, некоторые устройства ввода/вывода, соединенные между собой посредством локальной щины (рис. 2.15).

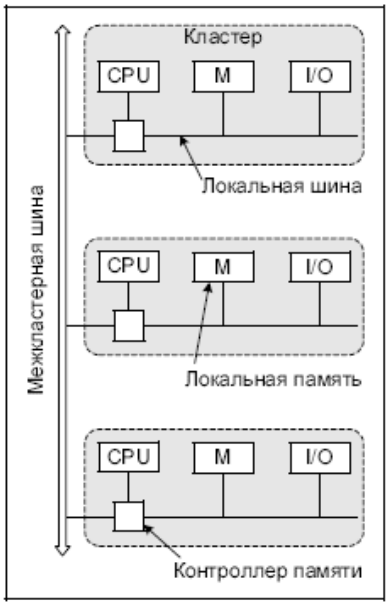


Рис. 2.15. Схема вычислительной системы Cm*

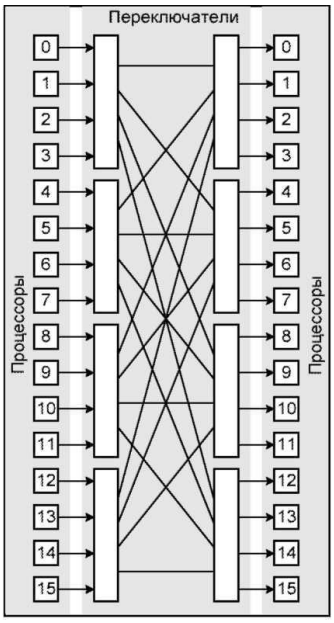


Рис. 2.16. Схема вычислительной системы BBN Butterfly

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 щину, в противном случае запрос для удаленного кластера отправляется через межкластерную щину. В таком режиме профамма, 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 профамма, 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 (Symmetric Multi Processors) 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 профаммиста 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. Эта проблема носит название проблемы согласования содержимого кэш-памяти (cache coherence problem, проблема когерентности кэшей).

Для решения проблемы согласования содержимого кэш-памяти (cache coherence problem, проблема когерентности кэшей) разработана специальная модификация NUMA-архитектуры — ccNUMA (cache coherent NUMA). Важно, что эта проблема решается и не ложится на плечи пользователей. Если обращение к памяти другого узла требует на 5—10% больше времени, чем обращение к своей памяти, то уровень "неоднородности" архитектуры NUMA не важен. 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 профамм 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 расщирения с архитектурой ccNUMA уже объединяют до 256 процессоров и больще.

15. Общая структура компьютера 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:

1. все процессы могут работать в едином адресном пространстве, адресуя любой байт памяти посредством обычных операций чтения/записи;
2. доступ к локальной памяти в системе будет идти немного быстрее, чем доступ к удаленной памяти;
3. проблемы возможного несоответствия данных, вызванные кэш-памятью процессоров, решены на уровне аппаратуры.

До 256 Гбайт оперативной памяти (планируется до 1 Тбайта).

Время такта приблизительно 1,3 **НС** (процессор PA-8700, 750 МГц), а пиковая производительность около 192 миллиардов арифметических операций в секунду.

Могут использоваться несколько типов микропроцессоров. Это процессоры семейства PA: PA-8600 и PA-8700, процессоры след.поколения с архитектурой IA-64 (компаний 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):

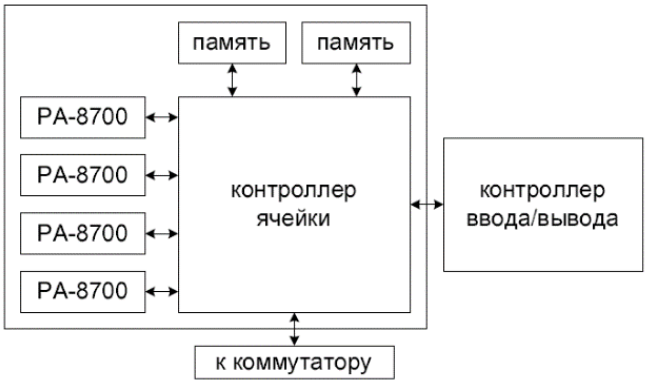


Рис. 3.12. Структура ячейки компьютера HP Superdome

- 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 AlphaSei-ver GS320 — QBB (Quad Building Block). Во всех системах в каждом элементе содержится по четыре процессора.

Контроллер ячейки. контроллер — состоит 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.

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 другим 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).

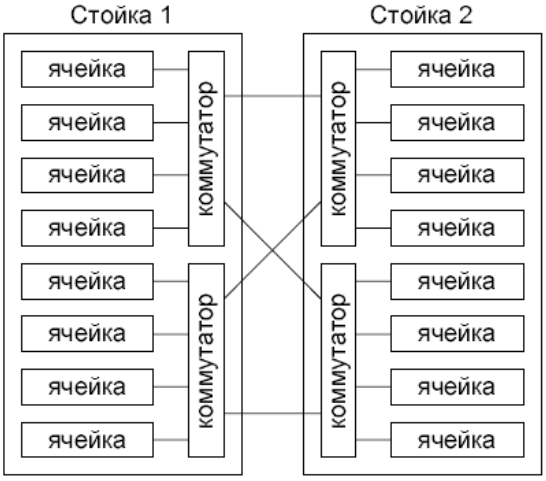


Рис. 3.13. Общая структура компьютера HP Superdome
Каждая стойка содержит по два восьмипортовых неблокирующих коммутатора. Все порты коммутаторов работают со скоростью 8 Гбайт/с. К каждому коммутатору подключаются четыре ячейки. Три порта коммутатора задействованы для связи с другими коммутаторами системы (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.
Ясно, что величина задержки зависит не только от взаимного расположения процессора и памяти, но и от числа процессоров. Не менее важным параметром является и характер вычислительной нагрузки. Например, задержка может меняться в зависимости от числа одновременно работающих приложений. В табл. 3.6 показана зависимость задержки от числа процессоров в двух характерных ситуациях. В первом случае работают однонитевые приложения, а во втором — многонитевая профамма. В отличие от первого случая, во втором случае появляются дополнительные затраты, необходимые для поддержания когерентности кэш-памяти процессоров. В обоих случаях приводятся усредненные показатели задержки. Предполагается, что запросы к памяти распределены равномерно.

Таблица 3.6. *Задержка в зависимости от числа процессоров и загрузки*

Число процессоров	Однонитевые программы, нс	Многонитевая программа, нс
4	174	235
8	208	266
16	228	296
32	261	336
64	275	360

Для рассмотренных видов нафузки, при сделанных выше предположениях о распределении запросов к памяти, показанные результаты являются очень неплохими. В самом деле, коэффициент увеличения задержки при переходе от минимальной 4-процессорной конфигурации к 64-процессорной составил лишь

1,6 раза. Это значит, что во многих случаях пользователь вправе надеяться на эффективную реализацию своих профамм, созданных им для традиционных SMP-компьютеров.

Компьютер HP Superdome имеет массу интересных особенностей. В частности, профаммно-аппаратная среда компьютера позволяет его настроить различным образом. Superdome может быть классическим единым компьютером с обшей памятью. Однако его можно сконфигурировать и таким образом, что он будет являться совокупностью независимых разделов (пPartitions), работаюших 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.

16. Причины уменьшения производительности компьютеров с общей памятью.

1. Закон Амдала;
2. неоднородность доступа к памяти;
3. конфликты при обращении к памяти;
4. необходимость обеспечения согласованности содержимого кэш-памяти;
5. сбалансированность вычислительной нагрузки;
6. реальная производительность отдельного процессора.

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.

17. Коммуникационные топологии компьютеров с распределенной памятью. Длина критического пути, связность, сложность.

Линейка, кольцо, звезда; двумерная решетка, двумерный тор, двоичный гиперкуб

Топологии сетей

•Звезда / Дерево
–Ethernet, InfiniBand
•Толстое дерево
–InfiniBand
•Тор, Гиперкуб
–SCI, InfiniBand
•Batterfly
–InfiniBand

Простейший вариант топологии связи показан на рис. 2.13, а, где все вычислительные узлы объединены в одну линейку. Каждый узел системы, кроме первого и последнего, имеет по одному непосредственному соседу справа и слева. Для построения системы из n узлов требуется n-1 соединение. Средняя длина пути между двумя узлами системы равна n/3. Можно уменьшить среднюю длину пути, если преобразовать линейку вычислительных узлов в **кольцо** (рис. 2.13, б). Добавив лишь одно дополнительное соединение первого узла с последним, мы на самом деле получили два дополнительных полезных свойства у новой топологии. Во-первых, средняя длина пути между двумя узлами сократилась с n/3 до n/6. Во-вторых, за счет того, что передача между любыми узлами уже может идти по двум независимым направлениям, увеличилась отказоустойчивость системы в целом. Пока все связи работоспособны, передача будет идти по кратчайшему пути. Но если нарушилась какая-либо одна связь, то возможна передача в противоположном направлении.

Схему сдваивания или блочные методы линейной алгебры на таких топологиях эффективно реализовать не просто, поскольку неправильное размещение процессов по процессорам приведет к потере большей части времени на коммуникации.

Например, использование схемы распределения работы между параллельными процессами, аналогичной схеме клиент-сервер, при которой один головной процесс раздает задания подчиненным процессам (схема мастер/рабочие, или master/slaves), хорошо соответствует топологии "**звезда**" (рис. 2.13, в).

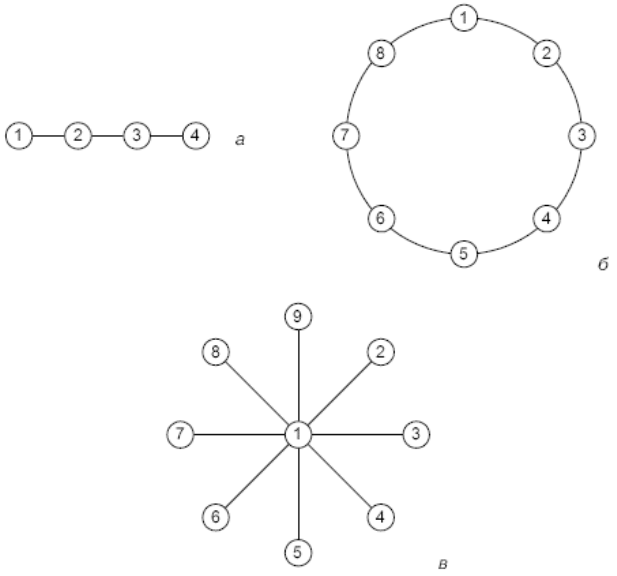


Рис. 2.13. Мультикомпьютерные системы с топологиями связи: *а* — линейка; *б* — кольцо; *в* — звезда

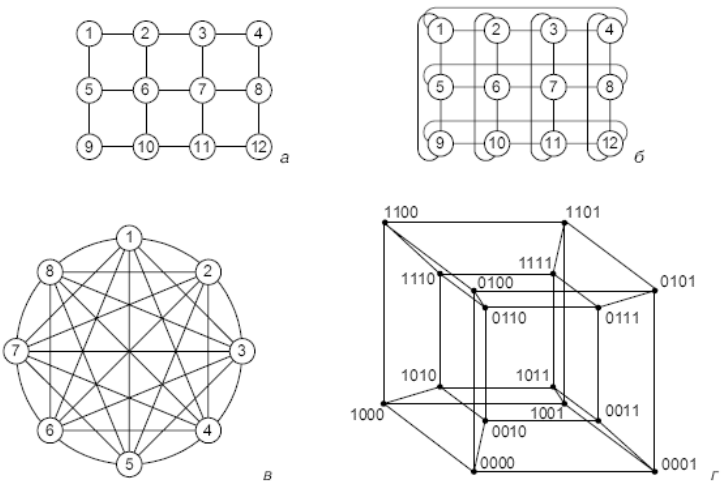


Рис. 2.14. Варианты топологий связи процессоров: *а* — решетка; *б* — 2-тор; *в* — полная связь; *г* — гиперкуб

Топология **двумерной решетки** (рис. 2.14, а) была использована в начале 90-х годов прошлого века при построении суперкомпьютера Intel Paragon на базе процессоров i860. В соответствии с топологией **двумерного тора** (рис. 2.14, б) могут быть соединены вычислительные узлы кластеров, использующих сеть SCI, предлагаемую компанией Dolphin Interconnect Solutions. Таким образом, в частности, построен один из кластеров НИВЦ МГУ. В настоящее время эта же компания Dolphin предлагает сетевые комплекты, позволяю-шие объединять узлы в трехмерный тор — чем меньше среднее расстояние между узлами, тем выше надежность. В этом смысле наилучшие показатели имеет топология, в которой каждый процессор имеет непосредственную связь со всеми остальными (рис. 2.14, в). Но о такой роскоши

современный уровень технологии даже мечтать не позволяет: п — 1 соединение у каждого узла при обшем числе связей п{п — 1)/2.
Иногда находятся исключительно интересные варианты, одним из которых является топология **двоичного гиперкуба** (рис. 2.14, г). В **й**-мерном пространстве рассмотрим лишь вершины единичного **й**-мерного куба, т. е. точки **(X|, X2,** в которых все координаты **X/** могут быть равны либо О, либо 1. В эти точки мы условно и поместим процессоры системы. Каждый процессор соединим с ближайшим непосредственным соседом вдоль каждого из **й** измерений. В результате получили **й**-мерный куб для системы из N =2" процессоров. Двумерный куб соответствует простому квадрату, а четырехмерный вариант условно изображен на рис. 2.14, г. В гиперкубе каждый процессор связан лишь с *\0g2N* непосредственными соседями, а не с j**V**, как в случае полной связности. Заметим, что при всей кажушейся замысловатости гиперкуб имеет массу полезных свойств. Например, для каждого процессора очень просто определить всех его соседей: они отличаются от него лишь значением какой-либо одной координаты **X/.** Каждая "фань" **й**-мерного гиперкуба является гиперкубом размерности й — 1. Максимальное расстояние между вершинами й-мерного гиперкуба равно п. Гиперкуб симметричен относительно своих узлов: из каждого узла система выглядит одинаковой и не существует узлов, которым необходима специальная обработка. Отметим и то, что многие алгоритмы по своей структуре прекрасно соответствуют такой взаимосвязи между процессорами. В частности, "неудобная" для других топологий схема сдваивания очень хорошо ложится на гиперкуб, используя на каждом шаге алгоритма гиперкуб на единицу меньшей размерности. (пример такой топологии: Cosmic Cube, построенный в 1983 году в Калифорнийском технологическом институте на основе микропроцессоров Intel 8086/8087.)

Длина пути - минимальное количество элементарных связей, которые нужно пройти для коммутации 2-х самых удаленных процессоров
Связность - количество элементарных связей, которые нужно удалить, чтобы схема распалась на 2 несвязные части
Сложность - общее количество необходимых элементарных связей

18. Общая структура компьютеров ссемейства CRAY XT: вычислительные узлы, процессорные элементы, коммуникационная сеть.

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 дан-ным, расположенным в памяти других процессоров, выполняется другими, более сложными способами.

CRAY T3D подключается к хост-компьютеру (главному или ведущему), роль которого, в ча-стности, может исполнять CRAY Y-MP C90. Вся предварительная обработка и подготовка про-грамм, 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 вычислительно-го узла, идентичны и могут работать независимо друг от друга.

Все узлы компьютера делятся на три группы. Когда пользователь подключается к компьютеру, то он попадает на **управляющие узлы.** Эти узлы работают в многопользовательском режиме, на этих же узлах выполняются однопроцессорные профаммы и работают командные файлы. **Узлы операционной системы** недоступны пользователям напрямую. Эти узлы поддерживают выполнение многих системных сервисных функций ОС, в частности, работу с файловой системой. **Вычислительные узлы** компьютера предназначены для выполнения профамм пользователя в монопольном режиме. При запуске профамме выделяется требуемое число узлов, которые за ней закрепляются вплоть до момента ее завершения. Гарантируется, что никакая профамма не сможет занять вычислительные узлы, на которых уже работает другая профамма. Число узлов каждого типа зависит от конфигурации системы. В частности, данные, взятые из двух реальных конфигураций Cray T3E, выглядят так: 24/16/576 или 7/5/260 (управляющие узлы/ узлы ОС/ вычислительные узлы).

Все марщрутизато-ры расположены в узлах трехмерной целочисленной прямоугольной рещет-ки и соединены между собой в соответствии с топологией трехмерного тора (рис. 3.14). Это означает, что каждый узел имеет щесть непосредственных соседей вне зависимости от того, где он расположен: внутри параллелепипеда, на ребре, на фани или в его верщине.

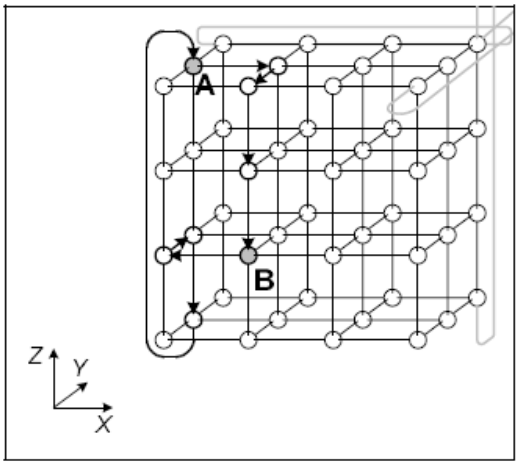


Рис. 3.14. Коммуникационная решетка компьютера Cray T3E

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 внут-реннюю кэш-память команд и кэш-память данных.

Объем локальной памяти ПЭ - 8 Мслов. Локальная память каждого процессорного элемен-та 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 собствен-ной локальной памяти.

Сетевой интерфейс формирует передачи перед посылкой через коммуникационную сеть другим вычислительным узлам или узлам ввода/вывода, а также принимает приходящие сооб-щения и распределяет их между двумя процессорными элементами узла.

Контроллер блочных передач - это контроллер асинхронного прямого доступа в память, ко-торый помогает перераспределять данные, расположенные в локальной памяти разных ПЭ ком-пьютера CRAY T3D, без прерывания работы самих ПЭ.

Коммуникационная сеть обеспечивает передачу информации между вычислительными уз-лами 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 эле-ментарная 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 ка-ждого пакета и могут осуществлять параллельный транзит данных по каждому из трех измере-ний *X, Y, Z.*

Нумерация вычислительных узлов.

Каждому ПЭ в системе присвоен уникальный ***физический номер***, определяющий его физи-ческое расположение, который и используется непосредственно аппаратурой.

Не обязательно все физические ПЭ принимают участие в формировании логической конфи-гурации компьютера. Например, 512-процессорная конфигурация компьютера CRAY T3D ре-ально содержит 520 физических ПЭ, 8 из которых находятся в резерве. Каждому физическому ПЭ присваивается ***логический номер***, определяющий его расположение в логической конфигу-рации компьютера, которая уже и образует трехмерный тор.

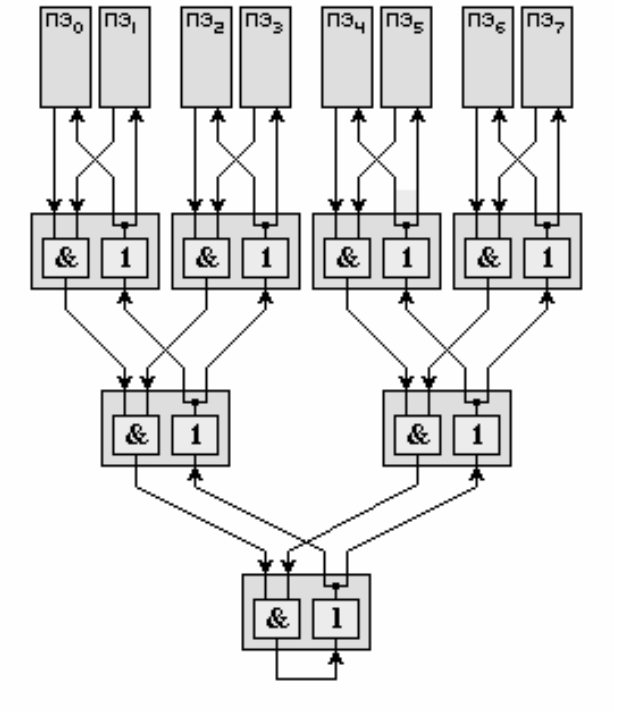
Каждой программе пользователя из трехмерной решетки вычислительных узлов выделяется отдельный раздел, имеющий форму прямоугольного параллелепипеда, на котором работает только данная программа (не считая компонент ОС). Для последовательной нумерации ПЭ, вы-деленных пользователю, вводится ***виртуальная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

19. Общая структура компьютеров семейства CRAY XT: аппаратная поддержка синхронизации параллельных процессов.

Для поддержки синхронизации процессорных элементов предусмотрена аппаратная реали-зация 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 ра-боту дальше.

В схемах поддержки каждого ПЭ предусмотрены два 8-ми разрядных регистра, причем каждый разряд регистров соединен со своей независимой цепью реа-лизации барьера (всего 16 независимых цепей). Каждая цепь строится на основе схем AND и ДУБЛИРОВА-НИЕ (1-2). До барьера соответствующие разряды на всех ПЭ обнуляются, а как только процесс на ПЭ дохо-дит до барьера, то записывает в свой разряд единицу. На выходе схемы AND появляется единица только в том случае, когда на обоих входах выставлены 1. Уст-ройство 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 распростра-няется всем ПЭ, сигнализируя о некотором событии на ПЭ. Это исключительно полезно, например, в задачах поиска.



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

Если говорить кратко, то вычислительный кластер есть совокупность компьютеров, объединенных в рамках некоторой сети для решения одной задачи (рис. 3.16). В качестве вычислительных узлов обычно используются доступные на рынке однопроцессорные компьютеры, двух- или четырех-процессорные SMP-серверы. Каждый узел работает под управлением своей копии операционной системы, в качестве которой чаше всего используются стандартные ОС: Linux, Windows NT, Solaris и т. п. Состав и мошность узлов могут меняться даже в рамках одного кластера, что дает возможность создавать неоднородные системы. Выбор конкретной коммуникационной среды определяется многими факторами: особенностями класса решаемых задач.

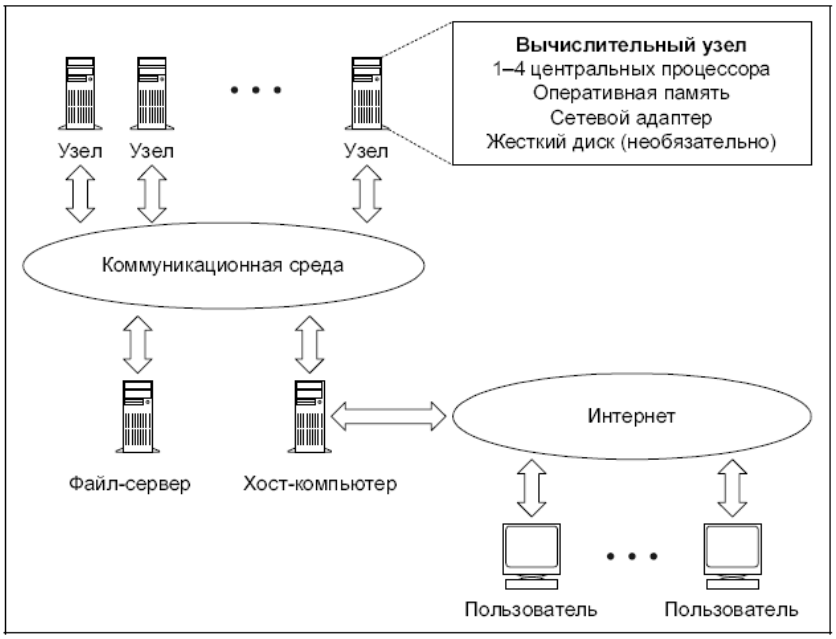


Рис. 3.16. Общая схема вычислительного кластера

Способы построения:

Рассматривая крайние точки, кластером можно считать как пару ПК, связанных локальной 10-мегабитной Ethernet-сетью, так и вычислительную систему, создаваемую в рамках проекта Cplant в Национальной лаборатории Sandia: 1400 рабочих станций на базе процессоров Alpha объединены высокоскоростной сетью Myrinet.

Различных вариантов построения кластеров очень много. Одно из существенных различий лежит в используемой сетевой технологии, выбор которой определяется, прежде всего, классом рещаемых задач. Первоначально BeowLilf-кластеры строились на базе обычной 10-мегабитной сети Ethernet. Сегодня часто используется сеть Fast Ethernet, как правило, на базе коммутаторов. Основное достоинство такого рещения — это низкая стоимость. Вместе с тем, больщие накладные расходы на передачу сообщений в рамках Fast Ethernet приводят к серьезным офаничениям на спекф задач, эффективно рещаемых на таких кластерах. Если от кластера требуется больщая универсальность, то нужно переходить на другие, более производительные коммуникационные технологии. Исходя из соображений стоимости, производительности и масштабируемости, разработчики кластерных систем делают выбор между Fast Ethernet, Gigabit Ethernet, SCI, Myrinet, cLAN, Sei-verNet и рядом других сетевых технологий.

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).

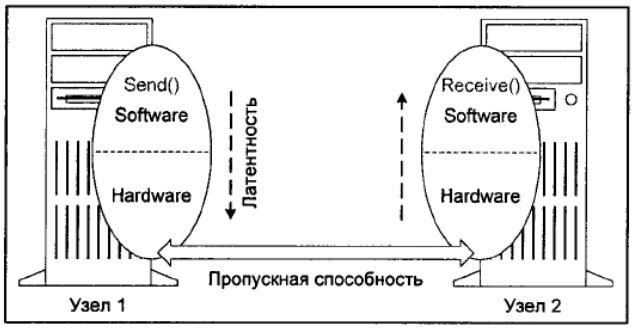


Рис. 3.19. Латентность и пропускная способность коммуникационной среды

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.

Предположим, что на двух процессорах (узлах) вычислительной системы работают два процесса, между которыми с помощью сети (другой коммуникационной среды) пересылаются сообщения. В передаче информации, помимо аппаратных устройств, участвует и программное обеспечение, например протокольный стэк (встроенный в ОС) и реализация интерфейса передачи сообщений MPI. Какими характеристиками определяется эффективность передачи информации между процессами параллельного приложения?

Основными характеристиками быстродействия сети являются **латентность** (latency) и **пропускная способность** (bandwidth).

Под **пропускной способностью** R сети будем понимать количество информации, передаваемой между узлами сети в единицу времени (байт в секунду). Очевидно, что реальная пропускная способность снижается программным обеспечением за счет передачи разного рода служебной информации.

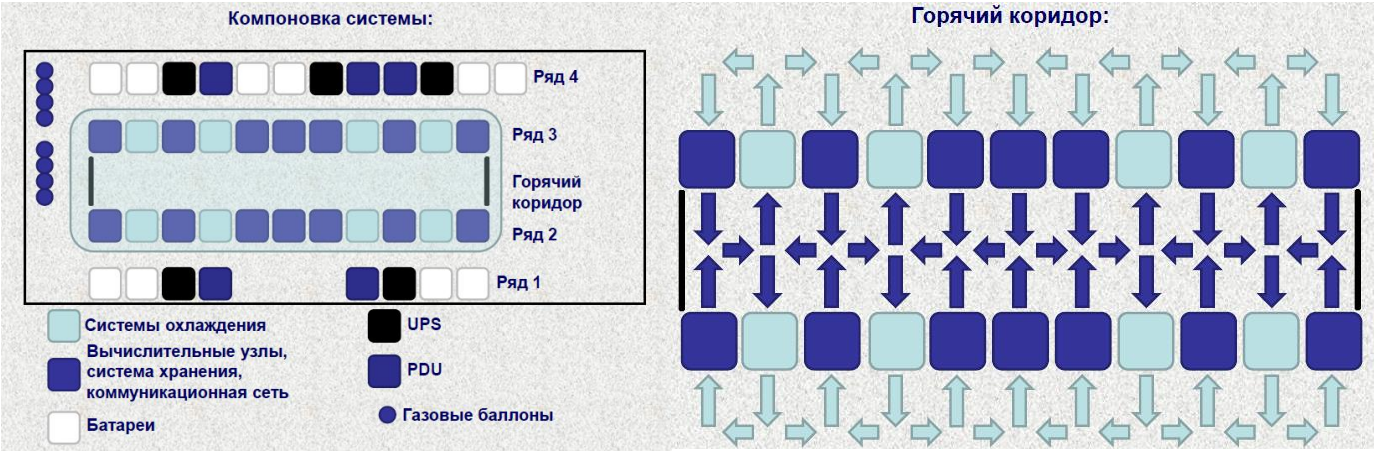
Латентностью (задержкой) называется время, затрачиваемое программным обеспечением и устройствами сети на подготовку к передаче информации по данному каналу. **Полная латентность** складывается из программной и аппаратной составляющих.

Различают следующие виды пропускной способности сети:

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

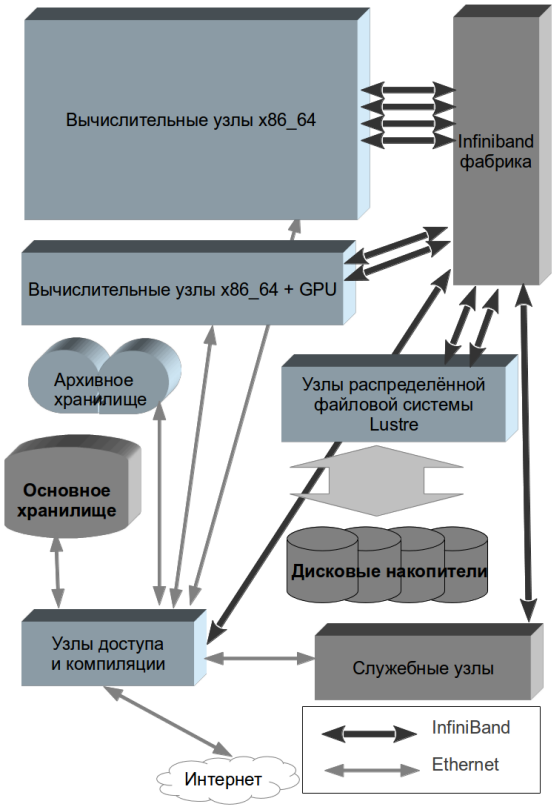
21. Архитектура суперкомпьютеров СКИФ МГУ «Чебышев» и «Ломоносов».
Суперкомпьютер «Ломоносов», 1.7 PFlop/s
Суперкомпьютер СКИФ МГУ «Чебышёв»
•1 место в 8-ой редакции списка Топ50 наиболее мощных компьютеров СНГ (март 2008 года)
•36 место в 31-ой редакции списка Top500 наиболее мощных компьютеров мира (июнь 2008 года)
•Пиковая производительность: 60 TFlop/s
•Производительность на Linpack: 47.32 TFlop/s (79% пиковой), матрица 740000x740000
•625 вычислительных узлов, 1250 процессоров, 5000 процессорных ядер
•42 стойки: 14 вычислительных, 28 инфраструктурных
•Помещение 98 м2
•Общий вес оборудования: более 30 тонн
•Энергопотребление вычислительной части 330 КВт, всего комплекса в пике до 720 КВт
•Система бесперебойного электропитания
•10 минут автономной работы
•Система охлаждения
•Звукоизоляция
•Система автоматического газового пожаротушения

Вычислительные узлы:
•Процессоры:
1250 Intel E5472 3.0 ГГц Harpertown
•Блэйд-шасси T-Blade («Т-Платформы»)
Форм-фактор 5 U
До 10 вычислительных узлов
•Оперативная память:
529 x 8 ГБ, бездисковые
64 x 8 ГБ, 160 ГБ HDD
32 x 16 ГБ, 160 ГБ HDD
8 x 32 ГБ, 160 ГБ HDD
Коммуникационная сеть:
•DDR InfiniBand
Mellanox MT25418 NIC
FatTree
SilverStorm 9120 – базовые коммутаторы
Flextronix F-X430046 – листовые коммутаторы
•Характеристики
1.3 – 1.95 μs латентность
1.7 ГБ/с пропускная способность
Вспомогательные сети:
•Gigabit Ethernet: коммутаторы Force10 C300 и Force10 S2410
•Управляющая сеть ServNet + IPMI **Хранилище данных:**
•60 ТБ распределённое отказоустойчивое сетевое хранилище T-Platforms ReadyStorage ActiveScale Cluster
•15 ТБ локальных дисков на узлах
•Ленточное хранилище Quantum Scalar i500 **Система охлаждения**
•8 кондиционеров APC InfraStruXure ACR502, уровень резервирования N+2
•Холодильные машины Liebert-Hiross SLH 023, одновременно работают 2 из 3 **Горячий коридор:**
•Меньший объём охлаждаемой части помещения
•Более тесная компоновка
•Встречные воздушные потоки



Система пожаротушения:
•Возможность ручного отключения всего комплекса
•Инертный газ
•3 месяца тестирования на ложные срабатывания
•При входе в помещение автоматическая система отключается
Электрическое оборудование
•1-ый и 4-ый ряды стоек
•PDU: APC AP9565
•UPS: APC Symmetra PX
•Мониторинг: ISX Manager
•Уровень резервирования N+1

Суперкомпьютер «Ломоносов»
Пиковая производительность
Производительность (Linpack)
Эффективность
Вычислительных узлов (Intel)
Вычислительных узлов (ГПУ)
Вычислительных узлов (PowerXCell)
Процессоры Intel Xeon 5570, 5670
NVIDIA Tesla X2070
Число процессорных ядер (x86)
Число процессорных ядер (ГПУ)
Оперативная память
Коммуникационная сеть
Система хранения данных
Операционная система
Занимаемая площадь (вычислитель)
Энергопотребление (вычислитель) 1700.21 TFlop/s 901.90 TFlop/s 53% 5 104 1 065 30 12 346 2 130 52 168 954 240 92 ТБайт QDR Infiniband / 10 GE 1.75 ПБайт, Lustre, NFS, … Clusrtx T-Platforms Edition 252 м2 2.8 МВт



Всего в системе охлаждения 10 т гликоля и 40 т воды
Общая длина кабелей более 80 км
Вес оборудования машзала – 57 т, СБЭ – 92 т

22. Топология коммуникационной сети «толстое дерево» (fat tree) на примере реализации в суперкомпьютерах СКИФ МГУ «Чебышёв» или «Ломоносов».

Схема построения Fat Tree в суперкомпьютере СКИФ МГУ «Чебышёв»

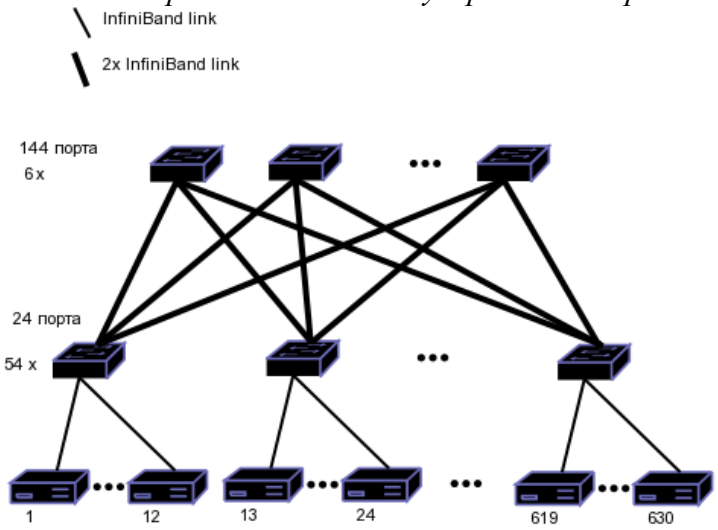
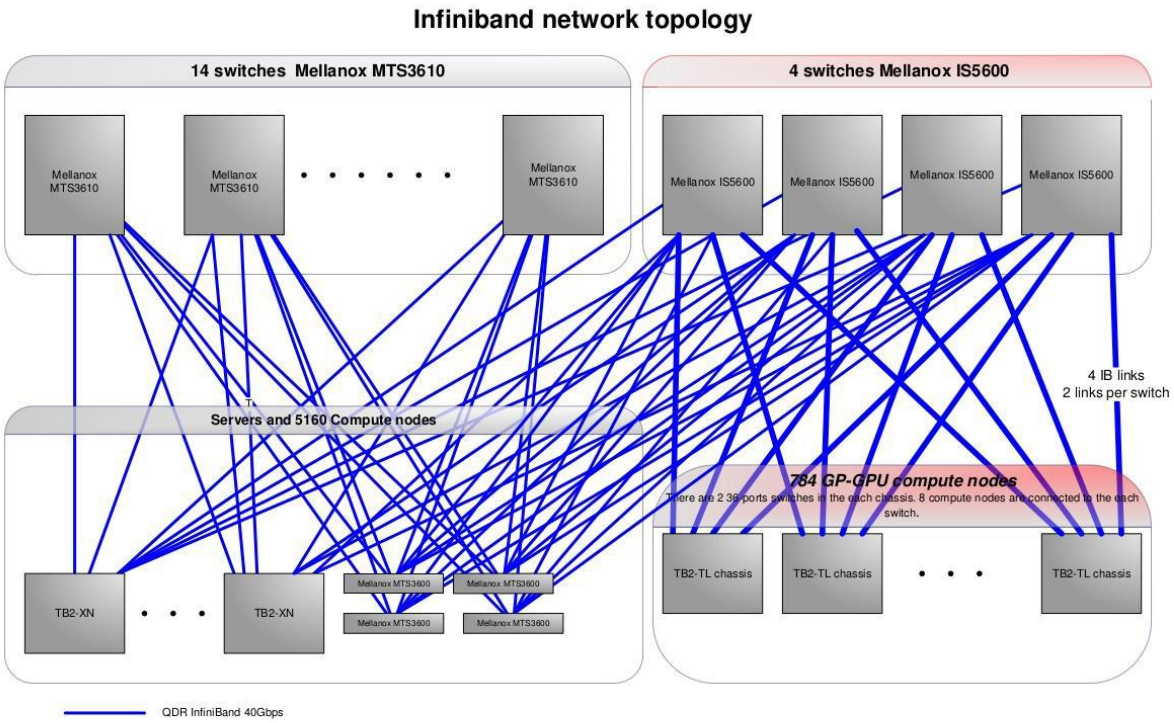


Схема построения Fat Tree в суперкомпьютере «Ломоносов»



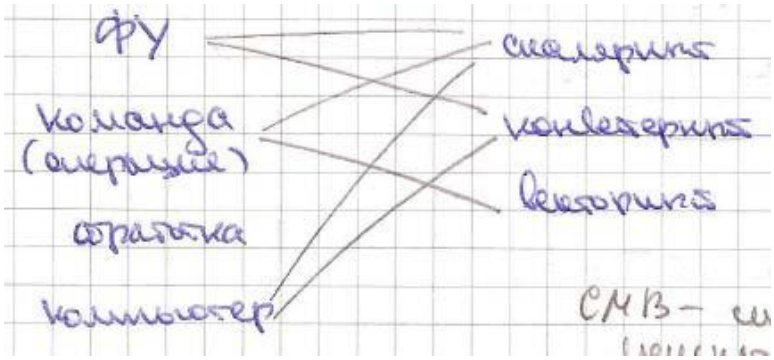
23. Причины уменьшения производительности компьютеров с распределённой памятью.

Из 6-лекции:
1. Закон Амдала
2. Латентность передачи по сети
3. Пропускная способность каналов передачи данных
4. Особенности использования SMP-узлов
5. Балансировка вычислительной нагрузки
6. Возможность асинхронного счета и передачи данных
7. Особенности топологии коммуникационной сети
8. Производительность отдельных процессоров
9. ...

Так как в программе всегда присутствует инициализация, ввод/вывод и некоторые сугубо последовательные действия, то недооценивать данный фактор никак нельзя - практически вся про-грамма должна исполняться в параллельном режиме, что можно обеспечить только после анали-за всей (!) программы.
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.
Для достижения эффективной параллельной обработки необходимо добиться ***равномерной загрузки всех процессоров***. Если равномерности нет, то часть процессоров неизбежно будет про-стаивать, 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 до-биться хорошей равномерности не так просто.
Если один процессор должен вычислить некоторые данные, которые нужны другому про-цессору, 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 выпол-нить 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 произво-дительность одного процессора***. Разные модели микропроцессоров могут поддерживать не-сколько уровней кэш-памяти, иметь специализированные функциональные устройства, регист-ровую 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.

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

Функциональное устройство:
• *скалярное;*
• *конвейерное (операция делится на несколько микроопераций, количество микроопераций определяет число ступеней конвейера).*
Команда:
• *скалярная (все аргументы - скаляры);*
• *векторная (например, сложить все элементы массива x с числом b).*
Компьютер:
• *скалярный;*
• *векторный;*
• *конвейерный.*



Любая вычислительная система есть работающая во времени совокупность функциональных устройств (ФУ). Назовем ФУ **простым**, если никакая последующая операция не может начать выполняться раньше, чем закончится предыдущая (напр., не конвейерные сумматоры или умножители). В отличие от простого ФУ, **конвейерное** ФУ распределяет свое оборудование для одновременной реализации нескольких операций.

Из **структуры компьютера CRAY Y-MP C90**
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 пла-вающей 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 вели-чины. Предназначены для выполнения как векторных, так и скалярных команд.
ФУ имеют различное число ступеней конвейера, но каждая ступень срабатывает за один такт, поэтому при полной загрузке все ФУ могут выдавать результат каждый такт.
Команды выбираются из ОП блоками и заносятся в буфера команд, откуда они затем выби-раются для исполнения. Если необходимой для исполнения команды нет в буферах команд, то происходит выборка очередного блока.
Команды имеют различный формат и могут занимать 1 пакет (16 разрядов), 2 пакета или 3 пакета (в одном слове 64 разряда, следовательно, в слове содержится 4 пакета). Максимальная длина программы на CRAY C90 равна 1 Гигаслову.

25. Векторизация программ, необходимые условия векторизации, препятствия для векторизации.

Вектор - упорядоченный набор однотипных данных, все элементы которого размещены в памяти компьютера с одинаковым смещением друг относительно друга.

Векторизация программы - процесс поиска подходящих фрагментов в программе и их замена векторными командами.

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 может быть в 10—15 раз выше скорости скалярной обработки.

Если весь фрагмент программы удалось заменить векторными командами, то говорят о его **полной векторизации**, иначе - **частичная векторизация** или **невозможность векторизации** фрагмента.

Пример векторизуемого фрагмента (на Cray C90) может выглядеть так:

```
 DO i = 1, n
     C(i) = A(i) + B(i)
END DO
```

Компилятор сгенерирует последовательность векторных команд: 1) загрузка векторов **A** и в из памяти в векторные регистры; 2) векторная операция сложения; 3) запись содержимого векторного регистра в память.

Необходимые условия векторизации:

1) наличие векторов-аргументов
2) над всеми элементами векторов должны выполняться одинаковые, независимые операции, для которых существуют аналогичные векторные команды в системе команд компьютера

Препятствия для векторизации:

1) зависимость по данным
2) отсутствие необходимых векторных команд в системе команд компьютера
3) Отсутствие регулярно расположенных векторов

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

4) Присутствие цикла, вложенного в данный - для реализации такого фрагмента нет соответствующих векторных команд
5) Вызов неизвестных подпрограмм и функций

```
Do i=1,n
      CALL SUBR(A, B)
End Do
```

Основные кандидаты для векторизации - самые внутренние циклы всех циклических конструкций программы, т.к. они задают перебор "одномерных" наборов данных, которые, в частности, могут быть векторами.

26. Общая структура компьютера CRAY C90. Параллелизм в архитектуре компьютера CRAY C90.

Cray C90 (самое начало 90-х) - это векторно-конвейерный компьютер, объединяющий в максимальной конфигурации **16 процессоров**, работающих над общей памятью. **Время такта** компьютера CRAY Y-MP C90 равно 4.1 нс, что соответствует **тактовой частоте** почти 250MHz.

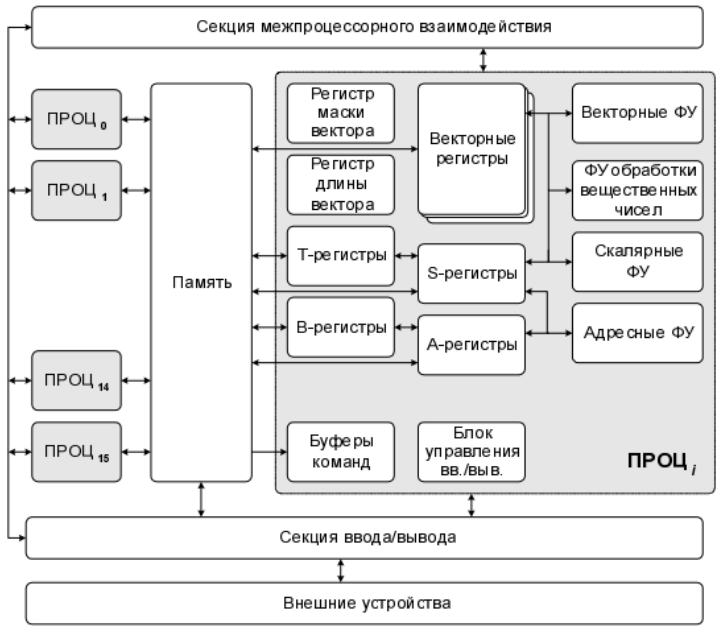


Рис. 3.8. Общая схема компьютера Cray C90

Его последователь Cray T90 имеет такую же структуру, отличаясь лишь некоторыми количественными характеристиками.

Все процессоры компьютера Cray C90 одинаковы и равноправны по отношению ко всем разделяемым ресурсам: памяти, секции ввода/вывода и секции межпроцессорного взаимодействия.

Оперативная память

Оперативная память Cray C90 разделяется всеми процессорами и секцией ввода/вывода. Каждое слово памяти состоит из 80-ти разрядов: 64 разряда для хранения данных и 16 вспомогательных разрядов для коррекции ошибок. Для увеличения скорости выборки данных вся память разделена на множество банков, которые могут работать одновременно.

Каждый процессор имеет доступ к оперативной памяти через четыре порта с пропускной способностью два слова за один такт каждый. Один из портов всегда связан с секцией ввода/вывода, и, по крайней мере, один из портов всегда выделен под операцию записи. Подобная архитектура хорошо подходит для выполнения векторных операций с не более чем двумя входными векторами.

В максимальной конфигурации реализовано расслоение памяти компьютера на 1024 банка: каждая из 8 секций разделена на 8 подсекций, а каждая подсекция на 16 банков. Последовательные адреса идут с чередованием по каждому из данных параметров:

адрес 0 — в 0-й секции, 0-й подсекции, 0-м банке;

адрес 1 — в 1-й секции, 0-й подсекции, 0-м банке;

адрес 2 — в 2-й секции, 0-й подсекции, 0-м банке;

…

адрес 8 — в 0-й секции, 1-й подсекции, 0-м банке;

адрес 9 — в 1-й секции, 1-й подсекции, 0-м банке;

…

адрес 63 — в 7-й секции, 7-й подсекции, 0-м банке;

адрес 64 — в 0-й секции, 0-й подсекции, 1-м банке;

адрес 65 — в 1-й секции, 0-й 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

Поддерживает три типа каналов для работы с внешними устройствами, которые различаются скоростью передачи данных:

- Low-speed (LOSP) channels — 6 Мбайт/с;
- High-speed (HISP) channels - 200 Мбайт/с;
- Veiy high-speed (VHISP) channels - 1800 Мбайт/с.

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.

Вычислительная секция 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 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:

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

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

3) 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.

4) 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.

Секция управления процессора.

Команды выбираются из ОП блоками и заносятся в буфера команд, откуда они затем выбираются для исполнения. Если необходимой для исполнения команды нет в буферах команд, то происходит выборка очередного блока. Команды имеют различный формат и могут занимать 1 пакет (16 разрядов), 2 пакета или 3 пакета (в одном слове 64 разряда, следовательно, в слове содержится 4 пакета). Максимальная длина программы на CRAY C90 равна 1 Гигаслову.

Параллелизм в архитектуре компьютера CRAY C90.

Основные особенности архитектуры, дающие заметный вклад в ускорение выполнения программ:

1) <u>Конвейеризация выполнения команд</u>

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

2) <u>Независимость функциональных устройств</u>

Большинство ФУ в CRAY C90 являются независимыми, поэтому несколько операций могут выполняться одновременно

3) <u>Векторная обработка</u>

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.

4) Зацепление функциональных устройств

Зацеплением векторных операций - использование регистра результатов векторной операции в качестве входного регистра для последующей векторной операции, т.е. выход сразу подается на вход.

5) Многопроцессорная обработка (16 процессоров)

Multiprogramming - выполнение нескольких независимых программ на различных процессорах.

Multitasking - выполнение одной программы на нескольких процессорах.

Пиковая производительность CRAY Y-MP C90 вычисляется так: функциональные устройства выдают два результата каждый такт (сдвоенные конвейеры), зацепление сложения и умножения дает четыре операции за такт, что составляет почти 1 Гфлопс (опер/с). Если работают все 16 процессоров, то 16 Гфлопс.

27. Причины уменьшения производительности векторно-конвейерных компьютеров.

1) **Закон Амдала**: время работы программы определяется ее самой медленной частью.

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. Также 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.

2) **секционирование длинных векторных операций** на порции по 128 элементов (векторные регистры имеют 128 64-разрядных слова)

3) **время начального разгона конвейера** (один раз в самом начале) относится к накладным расходам на организацию векторных операций на конвейерных функциональных устройствах. Очень короткие циклы выгоднее выполнять не в векторном режиме, а в скалярном, когда этих накладных расходов нет.

4) **Конфликты при обращении в память.** 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. 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.

5) **Ограниченная пропускная способность каналов передачи данных (memory bottleneck).** 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. Т.е. на операции с зацеплением, требующей три аргумента, часть времени будет потрачена впустую, т.к. на чтение смогут работать лишь два канала.

6) **Несбалансированность в использовании функциональных устройств.** 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.

7) **Отсутствие устройства деления в наборе функциональных устройств.**

8) **Перезагрузка буферов команд** происходит, 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.

Если предположить, что влияние каждого отдельного фактора в некоторой программе таково, что позволяет достичь 85% пиковой производительности, то их суммарное воздействие снизит реальную производительность менее, чем до 20% от пика!

28. Метакомпьютер и метакомпьютинг. Отличительные свойства распределенных вычислительных сред.

Любые вычислительные устройства можно считать параллельной вычислительной системой, если они работают одновременно и их можно использовать для решения одной задачи. Компьютер, состоящий из компьютеров, своего рода **метакомпьютер**. А процесс организации вычислений на такой вычислительной системе — **метакомпьютинг**.

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. В общем случае в качестве коммуникационной среды метакомпьютера может выступать любая сетевая технология. В данном случае для нас важен принцип, а возможностей для технической реализации сейчас существует предостаточно.

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.

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. Если потребовались вычислительные мощности для решения задачи, то вы подключаетесь к метакомпьютеру, выдаете задание и получаете результат.

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

1) 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.;

2) 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;

3) 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;

4) 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;

5) 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. 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.

---

29. Параллелизм на уровне машинных команд. Суперскалярность, VLIW, EPIC.

Параллелизм на уровне машинных команд (Instruction Level Parallelism) - скрытый от пользователя параллелизм. Пользователь не нуждается в специальном параллельном программировании. Кроме того, 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 (требует много ресурсов). 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). 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.

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.

VLIW в своей изначальной форме имело несколько недостатков, препятствующих массовому внедрению:

- Наборы инструкций VLIW не являются обратно совместимыми между различными поколениями процессоров. Если в более новом процессоре будет использоваться больше исполнительных устройств (например, АЛУ), то программы для нового процессора нельзя исполнить на старом, более узком процессоре (с меньшим количеством устройств).
- Задержки загрузки данных из иерархии памяти (кэшей, DRAM) не являются полностью предсказуемыми. Из-за этого статическое планирование инструкций загрузки и использования данных становятся крайне сложными.

Архитектура **EPIC** (Explicitly Parallel Instruction Computing), возникшая как совместная разработка фирм Intel и HP в 1997 году, является усовершенствованным вариантом технологии VLIW. Первым представителем данной стратегии стал микропроцессор Itanium компании Intel.

Особенностями архитектуры EPIC являются:

- большое количество регистров;
- масштабируемость архитектуры до большого количества функциональных блоков;
- явный параллелизм в машинном коде. Поиск зависимостей между командами осуществляет не процессор, а компилятор;
- предикация - команды из разных ветвей условного предложения снабжаются полями предикатов (полями условий) и запускаются параллельно;
- предварительная загрузка - данные из медленной основной памяти загружаются заранее.

Оба подхода имеют свои достоинства и недостатки. 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.

30. Производительность вычислительных систем, методы оценки и измерения.

Не существует наилучшего способа оценки производительности и последующего сравнения компьютеров.

В идеальной ситуации вычислять бы для каждой системы по некоторому закону одно число, которое и явилось его обобщенной характеристикой. Например, опираясь на параметры самого компьютера. С этой точки зрения, естественной характеристикой любого компьютера является его **пиковая производительность** - тот максимум, на который способен компьютер. Чтобы его вычислить, достаточно предположить, что все устройства компьютера работают в максимально производительном режиме. Если в процессоре есть два конвейерных устройства, то рассматривается режим, когда оба конвейера одновременно работают с максимальной нагрузкой. Если в компьютере есть 1000 таких процессоров, то пиковая производительность одного процессора просто умножается на 1000.

Иногда пиковую производительность компьютера называют его **теоретической производительностью**, т.к. производительность компьютера на любой реальной программе никогда не только не превысит этого порога, но и не достигнет его точно. Пиковая производительность компьютера вычисляется однозначно. Однако, на одних задачах удается получать 90% от пиковой производительности, а на других лишь 2%.

Традиционно используются два способа оценки пиковой производительности компьютера:

1) **MIPS (Million Instructions Per Second)** - число команд, выполняемых компьютером в единицу времени, которое говорит о скорости выполнения компьютером своих же инструкций. Но в какое число инструкций отобразится программа пользователя или отдельный ее оператор заранее не ясно.

2) За время выполнения одной операции умножения двух чисел в форме с плавающей запятой может выполниться десяток простых инструкций. Это послужило причиной введения другого способа измерения пиковой производительности: **Flops (Floating point operations per second) - число вещественных операций, выполняемых компьютером в единицу времени.** Операция a + b в тексте программы всегда соответствует одной операции сложения в коде программы. Пользователь знает вычислительную сложность своей программы, а на основе этой характеристики может получить нижнюю оценку времени ее выполнения.

Пиковая производительность получается при работе компьютера в идеальных условиях. Нет конфликтов при обращении к памяти, все берется с регистров, все устройства постоянно и равномерно загружены и т. п. Полезнее для пользователя **оценка эффективности программно-аппаратной среды на некоторых задачах или наборе задач. Синтетические (или искусственные) тесты** не имеют отношения к реальным приложениям; предназначены для создания стрессовой нагрузки на отдельные подсистемы компьютера**. Реальные тесты** выполняют реальные задачи над реальными данными.

Основные требования к тестам производительности:

- Непротиворечивость и понятность результатов.
- Легкость в использовании.
- Масштабируемость.
- Переносимость.
- Репрезентативность.
- Доступность теста и его исходного кода.
- Воспроизводимость.

LINPACK - программа предназначена для решения СЛАУ с плотной матрицей с выбором главного элемента по строке. Простой алгоритм, регулярные структуры данных, значительная вычислительная емкость, возможность получения показателей производительности, близких к пиковым, — все эти черты сделали тест исключительно популярным.

Этот тест рассматривается в трех вариантах:

1) Решается система с матрицей размера 100x100. Предоставляется уже готовый исходный текст, в который не разрешается вносить никаких изменений.

2) Размер матрицы: 1000x1000. Разрешается как внесение изменений в текст подпрограммы, реализующей заложенный авторами метод решения системы, так и изменение самого метода. Единственное ограничение — это использование стандартной вызывающей части теста, которая выполняет инициализацию матрицы, вызывает подпрограмму решения системы и проверяет корректность результатов. В этой же части вычисляется и показанная компьютером производительность, исходя из формулы для числа операций $2n^3/3 + 2n^2$ (n — это размер матрицы), вне зависимости от вычислительной сложности реально использованного алгоритма. В данном варианте достигались значения производительности, близкие к пиковой.

3) Разрешено использовать изложенную во втором варианте методику для матриц сколь угодно большого размера. Сколько есть оперативной памяти на всех вычислительных узлах системы, столько и

можно использовать. Для современных компьютеров размер матрицы уже перевалил за миллион, поэтому такой шаг был просто необходим.

В настоящее время LINPACK используется для формирования списка Top500 — пятисот самых мощных компьютеров мира **(www.top500.org).** Кроме пиковой производительности R_{peak} для каждой системы указывается величина R_{max}, равная производительности компьютера на тесте LINPACK с матрицей максимального для данного компьютера размера N_{max}. По значению R_{max} отсортирован весь список. Как показывает анализ представленных данных, величина R_{max} составляет 50—70% от значения R_{peak}. Интерес для анализа функционирования компьютера представляет и указанное в каждой строке значение $N_{1/2}$, показывающее, на матрице какого размера достигается половина производительности R_{max}.

Неплохим дополнением к тесту LINPACK является **набор тестов STREAM**. Этот набор содержит четыре небольших цикла, работающих с очень длинными векторами. Основное назначение тестов STREAM состоит в оценке сбалансированности скорости работы процессора и скорости доступа к памяти. Размеры массивов подбираются таким образом, чтобы ни один из них целиком не попадал в кэш-память.

Также тестовые программы могут браться из разных предметных областей и делать акцент на различных особенностях архитектуры. Часть программ может быть модельными приложениями, а часть может представлять реальные промышленные приложения. Одним из первых пакетов, построенных на таких принципах, стал набор из **24 Ливерморских циклов (The Livermore Fortran Kernels, LFK)**. В его состав вошли вычислительные ядра прикладных программ, используемых в Ливерморской национальной лаборатории США. Все включенные в пакет фрагменты программ являются простыми циклическими конструкциями, разобраться в которых не составляет никакого труда. Часть циклов представляют собой типичные вычислительные блоки типа скалярного произведения, рекурсий или вычисления значения полиномов. Другая часть содержит часто используемые конструкции языка Fortran. Некоторые циклы содержат фрагменты, эффективная компиляция которых может быть затруднена.

Пакет тестов PERFECT Club Benchmarks (Performance Evaluation for Cost-effective Transformations) появился в конце 80-х годов прошлого века. Он состоит из тринадцати программ, общим объемом более 50 000 строк на языке Fortran-77. Практически все программы являются реальными приложениями, взятыми из различных предметных областей: вычислительная гидродинамика, прогноз погоды, обработка сигналов, моделирование распространения вредных примесей в атмосфере, квантовая механика и др.

Наибольшую известность среди наборов тестов получил **пакет NAS Parallel Benchmarks (NPB).** В состав пакета входят две группы тестов, отражающих различные стороны реальных программ вычислительной гидродинамики. Пять программ представляют типичные вычислительные ядра, а три программы являются модельными приложениями. В описании тестов фигурируют пять различных классов решаемых задач, на практике чаше всего используются три старших класса: A, B и C. Обе версии тестов NPB в совокупности дают прекрасный материал для анализа эффективности функционирования вычислительных систем. Результаты первой версии дают профессионально полученную верхнюю оценку производительности для каждого теста. Эта работа была выполнена экспертами очень высокой квалификации. Данные второй версии пакета NPB показывают значения, которые будут характерны для, скажем так, "грамотного" пользователя.

Среди известных тестов, используемых сегодня для оценки производительности компьютеров, можно назвать **тесты SPEC**. В состав тестового набора входят тесты вычислительного характера, тесты скорости работы системы в качестве Web-серверов, почтовых серверов, тесты графических подсистем и многие другие. Состав тестов часто меняется.

В середине 2001 года появился **пакет программ SPEC OMPM2001**, предназначенный для измерения производительности параллельных систем с общей памятью, содержащих от 4 до 32 процессоров.

Никакое одно число или даже набор чисел не будут универсальной характеристикой производительности компьютера. Требуется **комплексное тестирование программно-аппаратной среды.** В зависимости от целей тестирования выделяют и используют следующие уровни:

□ **базовый уровень программного обеспечения:** тестирование эффективности работы операционной системы, компиляторов и систем программирования;

□ **базовый уровень аппаратуры:** определение скорости выполнения элементарных операций, скорости обмена между различными уровнями иерархии памяти и объемы доступной памяти на каждом уровне;

□ **уровень операций ввода/вывода:** анализ эффективности различных режимов чтения и записи данных при работе с внешними устройствами, определение скорости выполнения основных операций с файлами и целесообразности выполнения асинхронных операций ввода/вывода;

□ **базовый коммуникационный уровень:** определение параметров среды взаимодействия параллельных процессов, эффективности выполнения основных коммуникационных процедур и примитивов синхронизации;

□ **коммуникационный уровень приложений:** исследование эффективности отображения различных логических топологий процессов на коммуникационную среду, получение и анализ коммуникационных профилей характерных параллельных программ;

□ **уровень модельных приложений:** определение характеристик компьютера при выполнении простых программ различной структуры;

□ **уровень реальных приложений:** комплексная проверка характеристик компьютера при выполнении реальных программ.

□ **31. Технологии параллельного программирования: способы и подходы создания параллельных программ.**

Не нашла пока. Какие-то фото лекций есть в готовых ответах.

32. OpenMP: параллельная программа, нити, основные конструкции для организации параллельных и последовательных секций.

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, С и С++.

С точки зрения OpenMP, весь текст программы разбит на последовательные и параллельные области. В начальный момент времени порождается **нить-мастер** или **"основная" нить**, которая начинает выполнение программы со стартовой точки.

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 никак не влияет на изменение значения этой же локальной переменной в других нитях.

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. 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( ) Если не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме.

33. OpenMP: основные конструкции для распределения работы между нитями.

Все порожденные дополнительные нити исполняют один и тот же код. OpenMP предлагает несколько вариантов распределения работы между ними. Например, можно написать так:

```
IF( OMP_GET_THREAD_NUM() .EQ. 3 ) THEN
    < код для нити с номером 3 >
ELSE
    < код для всех остальных нитей >
ENDIF
```

Если в параллельной секции встретился оператор цикла, 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 ,

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) // 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.

34. 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 (директива SHARED),** либо **локальными (директива 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.

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 [()]

...

!$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.

35. MPI: параллельная программа, сообщение, группы, коммуникаторы, создание нового коммуникатора.

MPI (Message Passing Interface) - хорошо стандартизованный механизм для построения программ по модели обмена сообщениями. Существуют стандартные «привязки» MPI к языкам C и Fortran.

- Стандарт MPI 3.1 (4 июня 2015 года).
- Более 450 процедур.
- SPMD (Single Program Multiple Data) - основная модель параллельного программирования.
- Префикс MPI_
- #include “mpi.h” (mpif.h для языка Фортран)

Группа – упорядоченное множество процессов.

Коммуникатор – контекст обмена группы. В операциях обмена используются только коммуникаторы!

Коммуникаторы имеют в языке Си предопределённый тип MPI_Comm (в языке Фортран – тип INTEGER)

- MPI_COMM_WORLD – коммуникатор для всех процессов приложения.
- MPI_COMM_SELF – коммуникатор, включающий только текущий процесс.
- MPI_COMM_NULL – коммуникатор, не содержащий ни одного процесса.

Каждый процесс может одновременно входить в разные коммуникаторы.

Два основных атрибута процесса:

- коммуникатор (группа)
- номер процесса в коммуникаторе (группе).

Если коммуникатор содержит n процессов, то номера процессов в нём лежат в пределах от 0 до n–1.

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.

Атрибуты сообщения:

- номер процесса- отправителя
- номер процесса-получателя
- идентификатор сообщения - целое неотрицательное число в диапазоне от 0 до MPI_TAG_UB (не меньше 32767)
- коммуниктор

Для работы с атрибутами сообщений введена **структура MPI_Status.**

Возвращаемое значение (в Фортране – последним аргументом) для большинства процедур MPI - информация об успешности завершения. В случае успешного выполнения процедура вернёт значение MPI_SUCCESS, иначе - код ошибки.

- int MPI_Init(int *argc, char ***argv) - **Инициализация параллельной части** программы. Почти все другие процедуры MPI могут быть вызваны только после вызова MPI_Init. Инициализация параллельной части для каждого приложения должна выполняться только один раз.
- int MPI_Finalize(void) - **Завершение параллельной части** приложения. Все последующие обращения к большинству процедур MPI, в том числе к MPI_Init, запрещены. К моменту вызова MPI_Finalize каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.
- int MPI_Initialized(int *flag) - В аргументе flag возвращает 1, если вызвана после процедуры MPI_Init, и 0 в противном случае.
- int MPI_Finalized(int *flag) - В аргументе flag возвращает 1, если вызвана после процедуры MPI_Finalize , и 0 в противном случае. Эти процедуры можно вызвать до MPI_Init и после MPI_Finalize.

- MPI_COMM_DUP(comm, newcomm) - **дублирует существующий коммуникатор** comm вместе со связанными с ним значениями ключей. возвращает в аргументе newcomm новый коммуникатор с той же группой, любой скопированной кэшированной информацией, но с новым контекстом.
- comm коммуникатор (десткриптор)
- newcomm копия comm (дескриптор)
- MPI_COMM_CREATE(comm, group, newcomm)- **создает новый коммуникатор** newcomm с коммуникационной группой, определенной аргументом group и новым контекстом.
- int MPI_Comm_size(MPI_Comm comm, int *size) - В аргументе size возвращает **число параллельных процессов в коммуникаторе** comm. int MPI_Comm_rank(MPI_Comm comm, int *rank) В аргументе rank возвращает номер процесса в коммуникаторе comm в диапазоне от 0 до size-1.
- int MPI_Comm_free(MPI_Comm comm) - **Уничтожает группу,** ассоциированную с идентификатором comm, который после возвращения устанавливается в MPI_COMM_NULL.

36. MPI: синхронное и асинхронное взаимодействие процессов, различные виды операторов Send (Bsend, Ssend, Rsend)

Все процедуры передачи сообщений в MPI делятся на две группы. В одну группу входят процедуры, которые предназначены для взаимодействия только двух процессов программы. Такие операции называются индивидуальными или операциями типа точка-точка. Процедуры другой группы предполагают, что в операцию должны быть вовлечены все процессы некоторого коммуникатора. Такие операции называются коллективными.

Начнем описание процедур обмена сообщениями с обсуждения операций типа точка-точка. В таких взаимодействиях участвуют два процесса, причем один процесс является отправителем сообщения, а другой - получателем. Процесс-отправитель должен вызвать одну из процедур передачи данных и явно указать номер в некотором коммуникаторе процесса-получателя, а процесс-получатель должен вызвать одну из процедур приема с указанием того же коммуникатора, причем в некоторых случаях он может не знать точный номер процесса-отправителя в данном коммуникаторе.

Все процедуры данной группы, в свою очередь, так же делятся на два класса: процедуры с блокировкой(с синхронизацией) и без блокировки(асинхронные). Процедуры обмена с блокировкой приостанавливают работу процесса до выполнения некоторого условия, а возврат из асинхронных процедур происходит немедленно после инициализации соответствующей коммуникационной операции.

MPI предоставляет следующие модификации процедуры передачи данных с блокировкой MPI_SEND:

- MPR_BSEND – передача сообщения с буферизацией. Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в специальный буфер, и произойдет немедленный возврат из процедуры. Выполнение данной процедуры никак не зависит от соответствующего вызова процедуры приема сообщения. Процедура может вернуть код ошибки, если места под буфер недостаточно.
- MPI_SSEND – передача сообщения с синхронизацией. Выход из данной процедуры произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера посылки, но и гарантированном достижении процессом-получателем точки приема сообщения в программе. Использование передачи сообщений с синхронизацией может замедлить программу, но позволяет избежать наличия в системе большого количества не принятых буферизированных сообщений.
- MPI_RSEND – передача сообщения по готовности. Данной процедурой можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов процедуры, вообще говоря, является ошибочным и результат ее выполнения не определен. Гарантировать инициализацию приема сообщения перед вызовом процедуры MPI_RSEND можно с помощью операций, осуществляющих явную или неявную синхронизацию процессов. Во многих реализациях сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи данных.

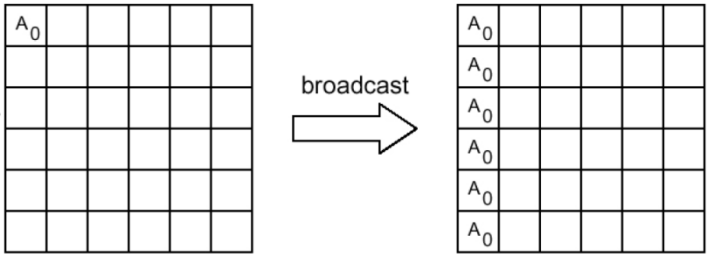
37. MPI: коллективные операции.

В операциях коллективного взаимодействия процессов *участвуют все процессы коммуникатора*! Как и для блокирующих процедур, возврат означает то, что разрешён свободный доступ к буферу приёма или посылки. Сообщения, вызванные коллективными операциями, не пересекутся с другими сообщениями. Вообще говоря, нельзя рассчитывать на синхронизацию процессов при помощи коллективных операций (кроме MPI_BARRIER). Если какой-то процесс завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммуникатора, ни даже того, что она ими начата. В коллективных операциях не используются идентификаторы сообщений (теги). Коллективные операции строго упорядочены согласно их появлению в тексте программы.

MPI_BARRIER(COMM, IERR)
INTEGER COMM, IERR

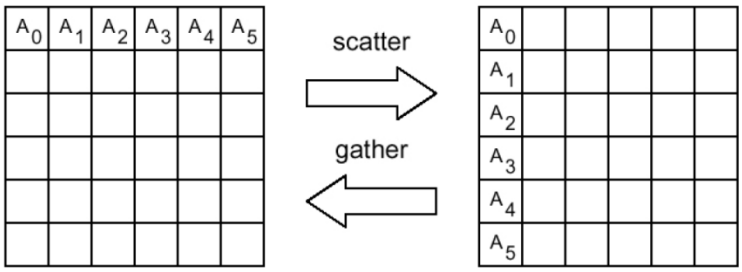
Процедура используется для барьерной синхронизации процессов. Работа процессов блокируется до тех пор, пока все оставшиеся процессы коммуникатора COMM не выполнят эту процедуру. Все процессы должны вызвать **MPI_Barrier**, хотя реально исполненные различными процессами коммуникатора вызовы могут быть расположены в разных местах программы.

MPI_BCAST(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
Рассылка **count** элементов данных типа **datatype** из массива **buf** от процесса **root** всем процессам данного коммуникатора **comm**, включая сам рассылающий процесс. Значения параметров **count**, **datatype**, **root** и **comm** должны быть одинаковыми у всех процессов.
По вертикали изображаются разные процессы, участвующие в коллективной операции, а по горизонтали – расположенные на них блоки данных.



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

Сборка **scount** элементов данных типа **stype** из массивов **sbuf** со всех процессов коммуникатора **comm** в буфер **rbuf** процесса **root**. Данные сохраняются в порядке возрастания номеров процессов.
На процессе **root** существенными являются значения всех параметров, а на остальных процессах - только значения параметров **sbuf**, **scount**, **stype**, **root** и **comm**. Значения параметров **root** и **comm** должны быть одинаковыми у всех процессов. Параметр **rcount** у процесса **root** обозначает число элементов типа **rtype**, принимаемых от каждого процесса.



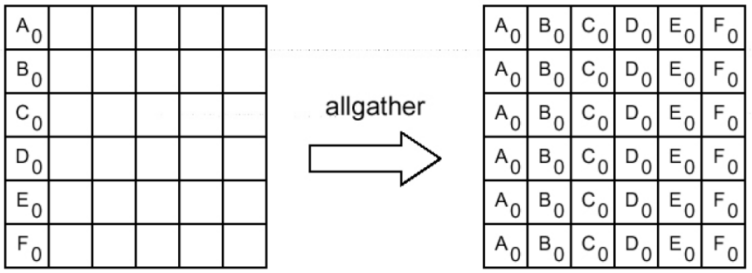
int MPI_Scatterv(void *sbuf, int *scounts, int *displs, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)

Рассылка различного количества данных из массива **sbuf.** Начало рассылаемых порций задает массив **displs.**
scounts – целочисленный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

displs – целочисленный массив, содержащий смещения относительно начала массива **sbuf** (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

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

Сборка данных из массивов **sbuf** со всех процессов коммуникатора **comm** в буфере **rbuf** каждого процесса. Данные сохраняются в порядке возрастания номеров процессов.

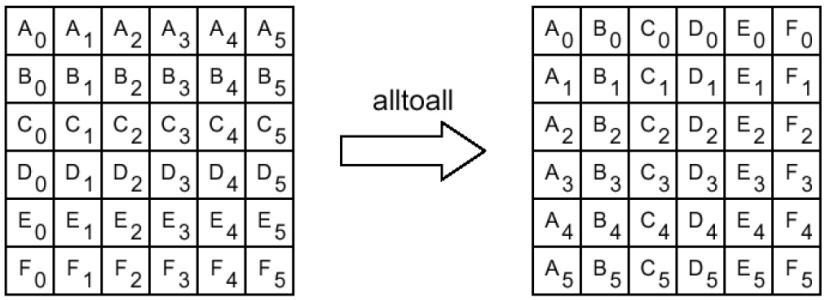


int MPI_Allgatherv(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int *rcounts, int *displs, MPI_Datatype rtype, MPI_Comm comm)

Сборка на всех процессах различного количества данных из **sbuf.** Порядок расположения задаёт массив **displs.**

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

Рассылка каждым процессом коммуникатора **comm** различных порций данных всем другим процессам. **j**-й блок массива **sbuf** процесса **i** попадает в **i**-й блок массива **rbuf** процесса **j**.



int MPI_Alltoallv(void* sbuf, int *scounts, int *sdispls, MPI_Datatype stype, void* rbuf, int *rcounts, int *rdispls, MPI_Datatype rtype, MPI_Comm comm)

Рассылка со всех процессов коммуникатора **comm** различного количества данных всем другим процессам**.** Размещение данных в буфере **sbuf** отсылающего процесса определяется массивом **sdispls**, а в буфере **rbuf** принимающего процесса – массивом **rdispls**.

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

Выполнение **count** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. Результат операции над **i**-ми элементами массивов **sbuf** получается в **i**-ом элементе массива **rbuf** процесса **root.**

Типы предопределённых глобальных операций:
MPI_MAX, MPI_MIN – максимальное и минимальное значения;
MPI_SUM, MPI_PROD – глобальная сумма и глобальное произведение;
MPI_LAND, MPI_LOR, MPI_LXOR – логические “И”, “ИЛИ”, искл. “ИЛИ”;
MPI_BAND, MPI_BOR, MPI_BXOR – побитовые “И”, “ИЛИ”, искл. “ИЛИ”.

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

Выполнение **count** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. Результат получается в массиве **rbuf** каждого процесса**.**

int MPI_Reduce_scatter(void *sbuf, void *rbuf, int *rcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Выполнение сумма по i-**rcounts(i)** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**. Сначала выполняются глобальные операции, затем результат рассылается по процессам**. i**-ый процесс получает **rcounts(i)** значений результата и помещает в массив **rbuf.**

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

Выполнение **count** независимых частичных глобальных операций **op** над соответствующими элементами массивов **sbuf**. **i**-ый процесс выполняет глобальную операцию над соответствующими элементами массива **sbuf** процессов **0…i** и помещает результат в массив **rbuf.** Окончательный результат глобальной операции получается в массиве **rbuf** последнего процесса.

int MPI_Op_create (MPI_User_function *func, int commute, MPI_Op *op)

Создание пользовательской глобальной операции. Если **commute=1**, то операция должна быть коммутативной и ассоциативной. Иначе порядок фиксируется по увеличению номеров процессов.

typedef void MPI_User_function (void *invec, void *inoutvec, int *len, MPI_Datatype type)

Интерфейс пользовательской функции.

int MPI_Op_free(MPI_Op *op)

Уничтожение пользовательской глобальной операции.

38. MPI: пересылка разнотипных данных, пересылка упакованных данных

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.
Для пересылки разнотипных данных можно использовать:

- Производные типы данных
- Упаковку данных

Производные типы данных создаются во время выполнения программы с помощью подпрограмм-конструкторов.
Создание типа состоит из двух этапов:
•Конструирование типа
•Регистрация типа

Производный тип данных характеризуется последовательностью базовых типов и набором значений смещения относительно начала буфера обмена.
Смещения могут быть как положительными, так и отрицательными, не требуется их упорядоченность. Таким образом, последовательность элементов данных в производном типе может отличаться от последовательности исходного типа, а один элемент данных может встречаться в конструируемом типе многократно.

int MPI_Type_contiguous(int count, MPI_Datatype type, MPI_Datatype *newtype)

Создание нового типа данных **newtype**, состоящего из **count** последовательно расположенных элементов базового типа данных **type**.

MPI_Type_contiguous(5, MPI_INT, &newtype) – тип данных для пересылки 5 расположенных подряд целых чисел

int MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype type, MPI_Datatype *newtype)

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** элементов после начала предыдущего.

Пример:

count=2;

blocklen=3;

stride=5;

MPI_Type_vector(count, blocklen, stride, MPI_DOUBLE, &newtype);

Создание нового типа данных (тип элемента, количество элементов от начала буфера):

{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 1), (**MPI_DOUBLE**, 2),

(**MPI_DOUBLE**, 5), (**MPI_DOUBLE**, 6), (**MPI_DOUBLE**, 7)}
---

int MPI_Type_create_hvector(int count, int blocklen, MPI_Aint stride, MPI_Datatype type, MPI_Datatype *newtype)

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** байт после начала предыдущего.

int MPI_Type_create_indexed_block(int count, int blocklen, int displs[], MPI_Datatype type, MPI_Datatype *newtype)

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Смещения блоков с начала буфера посылки в количестве элементов базового типа данных **type** задаются в массиве **displs**.

int MPI_Type_indexed(int count, int *blocklens, int *displs, MPI_Datatype type, MPI_Datatype *newtype)

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** элементов с начала буфера.

Создание нового типа данных для описания верхнетреугольной матрицы:

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

blocklens[i]=n-i;

displs[i]=(n+1)*i;

}

MPI_Type_indexed(n, blocklens, displs, MPI_DOUBLE, &newtype)

---

int MPI_Type_create_struct(int count, int *blocklens, MPI_Aint *displs, MPI_Datatype *types, MPI_Datatype *newtype)

Создание структурного типа данных из **count** блоков по **blocklens[i]** элементов типа **types[i]**. **i**-ый блок начинается через **displs[i]** байт с начала буфера.

---

blocklens[0]=3;

blocklens[1]=2;

types[0]=MPI_DOUBLE;

types[1]=MPI_CHAR;

displs[0]=0;

displs[1]=24;

MPI_Type_create_struct(2, blocklens, displs, types, &newtype);

Создание нового типа данных (тип элемента, количество байт от начала буфера):

{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 8), (**MPI_DOUBLE**, 16), (**MPI_CHAR**, 24), (**MPI_CHAR**, 25)}

---

int MPI_Type_create_subarray(int ndims, int sizes[], int subsizes[], int starts[], int order, MPI_Datatype type, MPI_Datatype *newtype)

newtype задаёт **ndims**–мерный подмассив исходного **ndims**–мерного массива. **sizes** задает размеры по каждому измерению исходного массива, **subsizes** – размеры по каждому измерению выделяемого подмассива.

starts задаёт стартовые координаты каждого измерения выделяемого подмассива в исходном массиве. Все массивы индексируются с 0. Задаваемые значения не должны выводить подмассив за пределы исходного

массива ни по одному из измерений. **order** задаёт порядок хранения элементов многомерного массива: **MPI_ORDER_C** (по строкам), **MPI_ORDER_FORTRAN** (по столбцам). **type** задаёт тип элементов массива.

int MPI_Type_commit(MPI_Datatype *datatype)

Регистрация созданного производного типа данных **datatype**. После регистрации этот тип данных можно использовать в операциях обмена.

int MPI_Type_free(MPI_Datatype *datatype)

Аннулирование производного типа данных **datatype**. **datatype** устанавливается в значение **MPI_DATATYPE_NULL**. Производные от **datatype** типы данных остаются. Предопределённые типы данных не могут быть аннулированы.

int MPI_Get_address(void *location, MPI_Aint *address)

Определение абсолютного байт-адреса **address** размещения массива **location** в оперативной памяти компьютера. Адрес отсчитывается от некоторого базового адреса, значение которого содержится в системной константе **MPI_BOTTOM**.

---

blocklens[0] = 1;

blocklens[1] = 1;

types[0] = MPI_DOUBLE;

types[1] = MPI_CHAR;

MPI_Get_address(dat1, &displs[0]);

MPI_Get_address(dat2, &displs[1]);

MPI_Type_create_struct(2, blocklens, displs, types, &newtype);

MPI_Type_commit(&newtype);

MPI_Send(MPI_BOTTOM, 1, newtype, dest, tag, MPI_COMM_WORLD);

---

int MPI_Type_size(MPI_Datatype datatype, int *size)

Определение размера типа **datatype** в байтах (объёма памяти, занимаемого одним элементом данного типа).

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)

Для элемента типа данных **datatype** определяет смещение от начала буфера данных нижней границы **lb** и диапазон **extent** (разницу между верхней и нижней границами) в байтах.

int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)

Упаковка **incount** элементов типа **datatype** из массива **inbuf** в массив **outbuf** со сдвигом **position** байт. **outbuf** должен содержать хотя бы **outsize** байт. Параметр **position** увеличивается на число байт, равное размеру записи. Параметр **comm** указывает на коммуникатор, в котором в дальнейшем будет пересылаться сообщение.
Для пересылки упакованных данных используется тип данных **MPI_PACKED**.

int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

Распаковка из массива **inbuf** со сдвигом **position** байт в массив **outbuf outcount** элементов типа **datatype**.

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

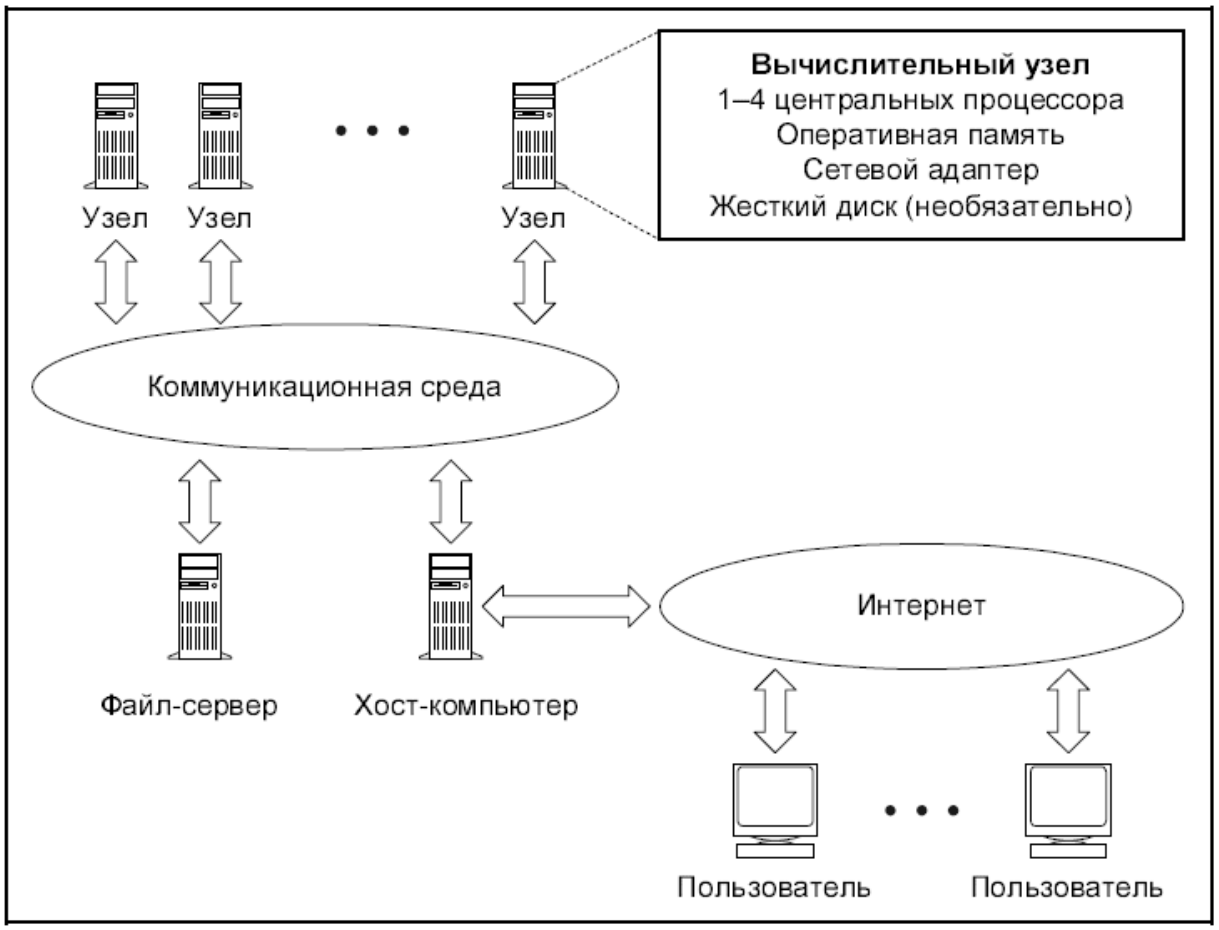
Определение необходимого объёма памяти (в байтах) для упаковки **incount** элементов типа **datatype**.

39. Варианты получения суперкомпьютера для своих вычислений. Их плюсы и минусы.

40. Основные компоненты вычислительного кластера, их назначение, варианты реализации.

Вычислительный кластер есть совокупность компьютеров, объединенных в рамках некоторой сети для решения одной задачи.

Что касается аппаратуры, на которой собирается кластер, основная составляющая здесь — межузловое соединение или внутренний кластерный интерконнект, обеспечивающий физическую и логическую связь серверов. На практике это внутренняя сеть Ethernet с продублированными соединениями. Ее назначение — во первых, передача пакетов, подтверждающих целостность системы (так называемых heartbeat), а во-вторых, при определенном дизайне или схеме, возникшей после возникновения неисправности, — обмен между узлами информационным трафиком, предназначенным для передачи вовне. Другие компоненты очевидны: узлы, на которых запущена ОС с кластерным ПО, дисковые хранилища, к которым имеют доступ узлы кластера. И наконец, общая сеть, через которую идет взаимодействие кластера с внешним миром. Программные компоненты обеспечивают управление работой кластерного приложения. Прежде всего это общая ОС (необязательно общая версия). В среде этой ОС работает ядро кластера — кластерное ПО. В качестве вычислительных узлов обычно используются доступные на рынке однопроцессорные компьютеры, двух- или четырех-процессорные SMP-серверы. Каждый узел работает под управлением своей копии операционной системы, в качестве которой чаше всего используются стандартные ОС: Linux, Windows NT, Solaris и т. п. Состав и мошность узлов могут меняться даже в рамках одного кластера, что дает возможность создавать неоднородные системы. Выбор конкретной коммуникационной среды определяется многими факторами: особенностями класса решаемых задач.



41. Параллельная реализация вычислительно сложных задач на примере задачи перемножения матриц.

Рассматривается еще один способ параллельных вычислений – задача перемножения матриц, но для четырехядерных процессоров с помощью технологии MPI.

Для параллельного решения задач на примере перемножения матриц с большим количеством элементов (больше 1000), требующие интенсивной работы с оперативной памятью справедливы утверждения:

1. Многоядерные процессоры не позволяют достичь теоретического ускорения, которое равно количеству ядер для большинства программ, написанных для последовательных систем, которые подверглись распараллеливанию.

2. Системы с распределенной памятью (кластеры для параллельных вычислений), несмотря на относительные медленные каналы связи между вычислительными узлами, позволяют для большинства простых программ достичь ускорения, соответствующее количеству процессоров в системе.

3. Для получения максимального эффекта от применения многоядерных процессоров требуется существенно усложнять программы, приспосабливая их для эффективного использования маленькой кэш памяти 1-го уровня. Это может привести к резкому возрастанию затрат на написание прикладных программ и сделать процесс программирования на многоядерных системах неэффективным.

42. Графовые модели программ, их взаимосвязь

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

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.

42. Графовые модели программ, их взаимосвязь.

информационная структура программы есть совокупность сведений о том, как отдельные элементы профаммы связаны между собой.

В основе изучения информационной структуры программ уже давно сформировались два подхода.

1) денотационный(опирается на исследование состояния памяти программ, на практике редок, поэтому не рассматриваем).
2) операционный. Исполнение (развитие) любой программы представляется в виде набора выполненных действий (операций), некоторым образом связанных между собой. Содержание действий, их вычислительная мощность и способ связи в рамках данного подхода не фиксируются. Они определяются в каждом конкретном случае по-своему в зависимости от целей исследования. Операционный подход в практических приложениях используется очень часто.

Он очевидным образом порождает класс **графовых** моделей программ. В этом классе каждая модель представляет граф, в котором вершины соответствуют каким-то множествам действий программы, а дуги (ребра) так или иначе связаны с отношениями между ними. Диапазон уровней сложности графовых моделей огромен. Отдельная вершина может представлять и программу целиком, и какие-то ее большие или малые фрагменты и даже отдельные элементарные операции. И на все это накладывается многообразие связей.

В любой программе можно выделить **два типа действий**, которые будем называть преобразователями и распознавателями. **Преобразователи** осуществляют переработку информации (операторы присваивания),

а **распознаватели** определяют последовательность срабатываний преобразователей в процессе работы программы (условные, разного рода переключатели и т. п. Основное их назначение состоит в выборе одной из нескольких возможных альтернатив дальнейшего следования). **Пусть множества данных операторов не пересекаются.**

Между действиями программы устанавливаются, как правило, два типа отношений:

1) Первый тип определяется фактом выполнения одного действия непосредственно за другим. Если какие-либо два действия находятся в этом отношении, то будем говорить, что между ними установлена связь по управлению или операционная связь.
2) Второй тип связан с использованием одним действием в качестве аргументов результатов выполнения других действий. Он определяет информационную связь.

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 - **граф управления**, управляющим графом, графом передач управления или просто графом программы(основное свойство- независимость от входных данных программы). Множества вершин и дуг для каждой программы фиксированы и образуют единственный граф. Этот граф задает одну из моделей программы. На рис. 6.1 он представлен для примера (6.1).

Теперь пусть 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 отличие от графа управления, в данном случае порядок непосредственного срабатывания операторов можно определить точно. Соединив вершины дугами передач управления, получим ориентированный **граф, носящий название операционно-логической истории программы**. Он 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 операционно-логической истории от входных данных зависит практически все: общее число вершин, количество вершин, соответствующих одному оператору, и даже набор присутствующих преобразователей и распознавателей. Граф управления свободен от подобных конкретностей.

Изменим графовую основу. Будем среди операторов принимать во внимание только преобразователи, а в качестве отношения между ними брать отношение информационной зависимости. Построим сначала граф,

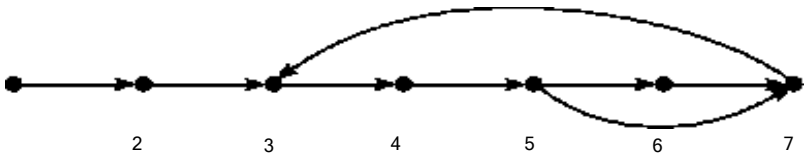


Рис. 6.1. Управляющий граф

в котором вершины соответствуют операторам-преобразователям. Две вершины соединим информационной дугой, если между какими-нибудь срабатываниями соответствующих операторов теоретически возможна информационная связь-**информационным графом программы** (не зависит от входных данных). В нем могут быть "лишние" дуги, которые не реализуются либо при конкретных входных данных, либо совместно с другими дугами.

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. Соединив вершины дугами передач информации, мы получим ориентированный **граф, называемый историей реализации программы.**

Мы имеем **4 основные графовые модели**. **Это — граф управления, информационный фаф, операционно-логическая история и история реализации.** Первые две модели не зависят от входных данных и строятся непосредственно по тексту программы. Две последние модели для своего построения формально требуют слежения за выполнением всех операторов. Понятно, что сложность построения моделей возрастает в порядке указанного перечисления. Первые две модели давно и успешно применяются на практике. Последние две модели до недавнего времени применялись, главным образом, в теоретических исследованиях, что связано, естественно, со сложностью их практического построения. Достоинством всех моделей является то, что они существуют для всех программ. На основе этих моделей можно строить

различные смешанные модели, в которых одни фрагменты программы представляются информационным графом или графом управления, а другие — той или иной историей.

43. Понятия информационной зависимости и информационной независимости. Примеры использования.

Трудности определения информационных связей между операциями по тексту программы привели к появлению большого числа работ, связанных с графовыми моделями иного типа, называемыми графами зависимостей. В этих моделях отношение информационной связи заменяется значительно более широким отношением зависимости. Именно, две операции или два оператора называются зависимыми, если при их выполнении имеют место обращения к одной и той же переменной (ячейке памяти). Так как информационная связь также реализуется через обращение к одной переменной, то ясно, что она есть частный случай зависимости. Очевидно, что устанавливать факт зависимости значительно проще, чем факт информационной связи. Во всех этих моделях мы, как правило, имеем дело только с операторами-преобразователями.

С 333-334 – примеры использования с рисунками.

Какое отношение выбрать для описания свойств программ?

Информационная структура – это основа анализа свойств и программ алгоритмов.

Информационная зависимость определяет критерий эквивалентности преобразований программ.

Информационная независимость определяет ресурс параллелизма программы.

44. Граф алгоритма. Критический путь графа алгоритма.

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 взаимно-однозначно 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 v. 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. Или предполагать, что все входные данные и результаты вводятся и выводятся через специальные устройства ввода/вывода. В этом случае вершины графа, отвечающие таким операциям, и только они не будут иметь соответственно входящие и выходящие дуги. Мы будем поступать в зависимости от обстоятельств. Построенный таким образом граф можно было бы назвать **графом информационной зависимости реализации алгоритма при фиксированных входных данных, или 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.
Это понятие требует некоторого пояснения. Граф алгоритма почти всегда зависит от входных данных. Даже если в программе отсутствуют условные операторы, он будет зависеть от размеров массивов, т. к. они определяют общее число выполняемых операций и, следовательно, общее число вершин графа. Так что в действительности граф алгоритма почти всегда есть параметризованный граф. От значений параметров зависит, конечно, не только число вершин, но и вся совокупность дуг. Если программа не имеет условных операторов, то как ее саму, так и описанный ею алгоритм будем называть детерминированными. В противном случае будем называть их недетерминированными. Имеется одно принципиальное отличие графа детерминированного алгоритма от графа недетерминированного алгоритма. Для **детерминированного алгоритм**а всегда существует взаимно-однозначное соответствие между всеми операциями описывающей его программы и всеми вершинами графа алгоритма. Для **недетерминированного алгоритма** взаимно-однозначного соответствия при всех значениях параметров, т. е. входных данных, нет. Можно лишь утверждать, что в этом случае при каждом наборе значений параметров существует взаимно-однозначное соответствие между каким-то подмножеством всех операций программы и вершинами графа. Разным значениям параметров могут соответствовать разные подмножества.

В дальнейшем, если не сделаны дополнительные оговорки, мы будем рассматривать детерминированные алгоритмы и программы..

Введенный в рассмотрение **граф алгоритма есть ориентированный ациклический мультиграф.** Его ацикличность следует из того, что в любых программах реализуются только явные вычисления и никакая величина не может определяться через саму себя. Даже если в программе имеются рекурсивные выражения, то это всего лишь удобная форма описания однотипных вычислений. При каждом обращении к рекурсии на самом деле реализуются разные операции. В общем случае граф алгоритма есть мультиграф, т. е. две вершины могут быть связаны несколькими дугами.
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.

Критический путь графа — путь максимальной длины в ориентированном ациклическом графе.

При аналитическом задании графа нахождение длины его критического пути как функции внешних параметров задачи является одной из важных задач при распараллеливании алгоритмов. При этом даже в случае, когда алгоритм относится к простому, например, линейному классу, заранее нельзя предугадать, к какому классу функций будет относиться длина критического пути.

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

Чтобы использовать обнаруженный в программе параллелизм, его нужно прежде всего каким-то образом записать. Обычно параллелизм описывают с помощью специальных циклов, а для этого надо программу преобразовывать. Остановимся сначала на общих принципах выполнения таких преобразований.

Будем называть два арифметических выражения эквивалентными, если их записи совпадают с точностью до обозначения переменных. Перенумеруем одинаковым образом входные переменные эквивалентных выражений. Рассмотрим две линейные программы. Предположим, что между их пространствами итераций установлено такое взаимно однозначное соответствие, при котором соответствующие точки задают срабатывания операторов с эквивалентными выражениями в правых частях. Подобные пространства итераций будем называть эквивалентными, а указанное соответствие — соответствием эквивалентности.
Пусть фиксировано правило, по которому для линейной программы однозначно строится какой-нибудь граф зависимостей. Возьмем две программы с эквивалентными пространствами итераций и предположим, что соответствие эквивалентности является изоморфным для графов. Кроме того, будем считать, что сопоставляемые при изоморфизме дуги графов зависимостей относятся к входным переменным с одними и теми же номерами. Такие программы будем называть эквивалентными по графам зависимостей, а сами графы — графами эквивалентности. Любое преобразование программы в эквивалентную ей программу будем называть эквивалентным.

Введенное понятие эквивалентности программ зависит от того, какие графы мы хотим принять во внимание при преобразовании программ. Все эквивалентные программы выполняют одно и то же множество операций, описанных арифметическими выражениями. Однако при последовательной реализации программ порядки выполнения операций могут различаться. В общем случае разные порядки могут приводить к разным результатам. При специальном выборе графов эквивалентности эквивалентные программы дают в одинаковых условиях реализации одни и те же результаты, включая всю совокупность ошибок округления. Одинаковые условия предполагают, в частности, что ошибка округления при выполнении любой операции на конкретном компьютере определяется только входными данными операции. Этому требованию удовлетворяют все используемые на практике компьютеры. Множества используемых переменных в эквивалентных программах могут быть организованы в массивы по-разному и могут различаться по их числу. В частности, по-разному могут быть организованы переменные, через которые задаются входные данные и которые формируют результаты.

В реальных условиях приходится выполнять также неэквивалентные преобразования. Типичным примером является замена одного оператора последовательностью операторов, реализующих в точных вычислениях ту же самую операцию. С формальной точки зрения преобразование не является эквивалентным, т. к. меняется пространство итераций. В практическом отношении такая замена из-за влияния ошибок округления чаще всего приводит к изменению результатов вычислений, что также не дает основание считать подобное преобразование эквивалентным.

Элементарные преобразования циклов:

1. **Перестановка циклов**

   Преобразование состоит в перестановке местами каких-либо двух циклов в тесно вложенном гнезде циклов. Не ограничивая существенно общность, можно считать, что первый (самый внешний) цикл переставляется с ***к-ым***

2. **Распределение цикла**

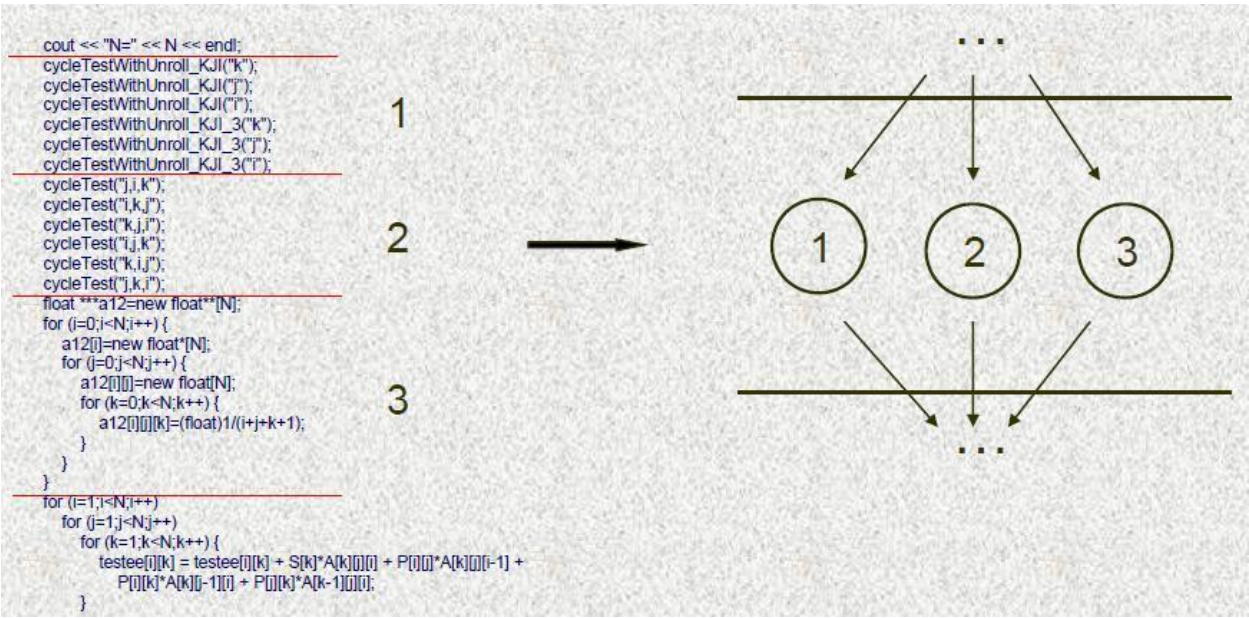
Пусть задан некоторый цикл. Построим для него граф зависимостей. Допустим, что после переупорядочивания в теле цикла его можно представить в виде двух подряд идущих фрагментов, и нет ни одной дуги, идущей из опорных пространств второго фрагмента в опорные пространства первого фрагмента. Тогда цикл можно представить в виде двух подряд идущих циклических конструкций. Самые внешние циклы у них совпадают. Телом первого цикла является первый фрагмент, телом второго — второй фрагмент.

Утверждение: для того чтобы можно было выполнить распределение цикла необходимо и достаточно, чтобы распределяемые части находились в разных компонентах сильной связности информационного графа тела данного цикла.

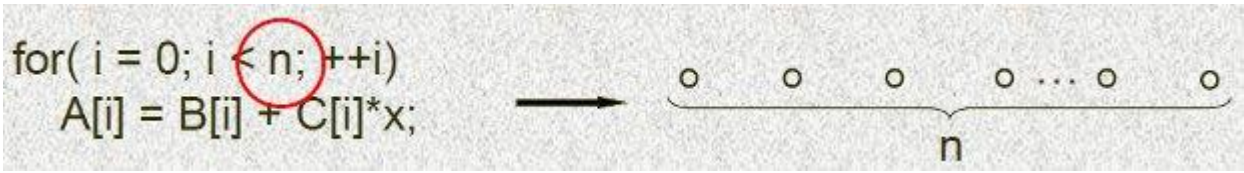
46. Виды параллелизма: конечный, массовый, координатный, скошенный.

Параллелизм:

1) Конечный определяется информационной независимостью некоторых фрагментов в тексте программы.



2) Массовый определяется информационной независимостью итераций циклов программы.



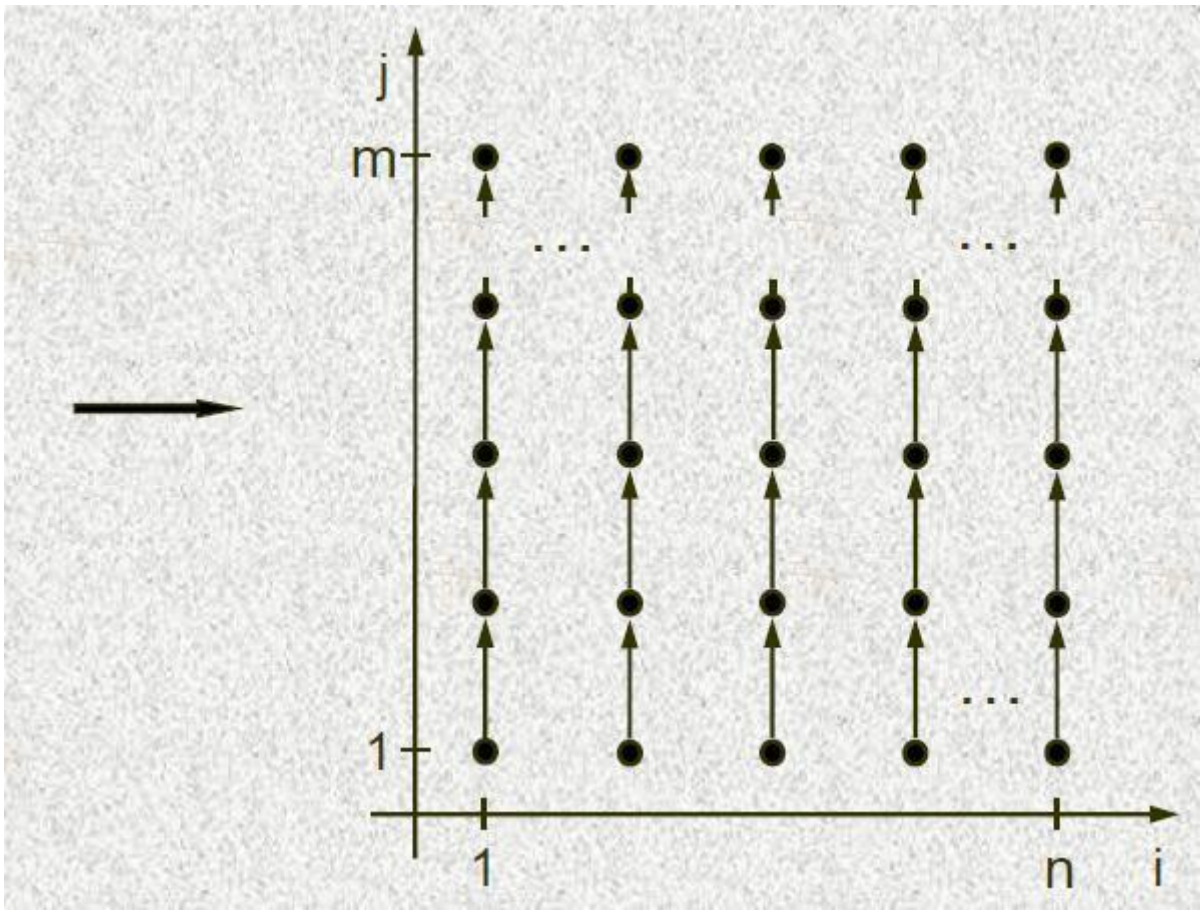
А) Координатный

Утверждение: для того чтобы цикл был параллельным необходимо и достаточно, чтобы для любой тройки графа алгоритма данного цикла включение $\Delta_i \subset G_i$ было верным, где

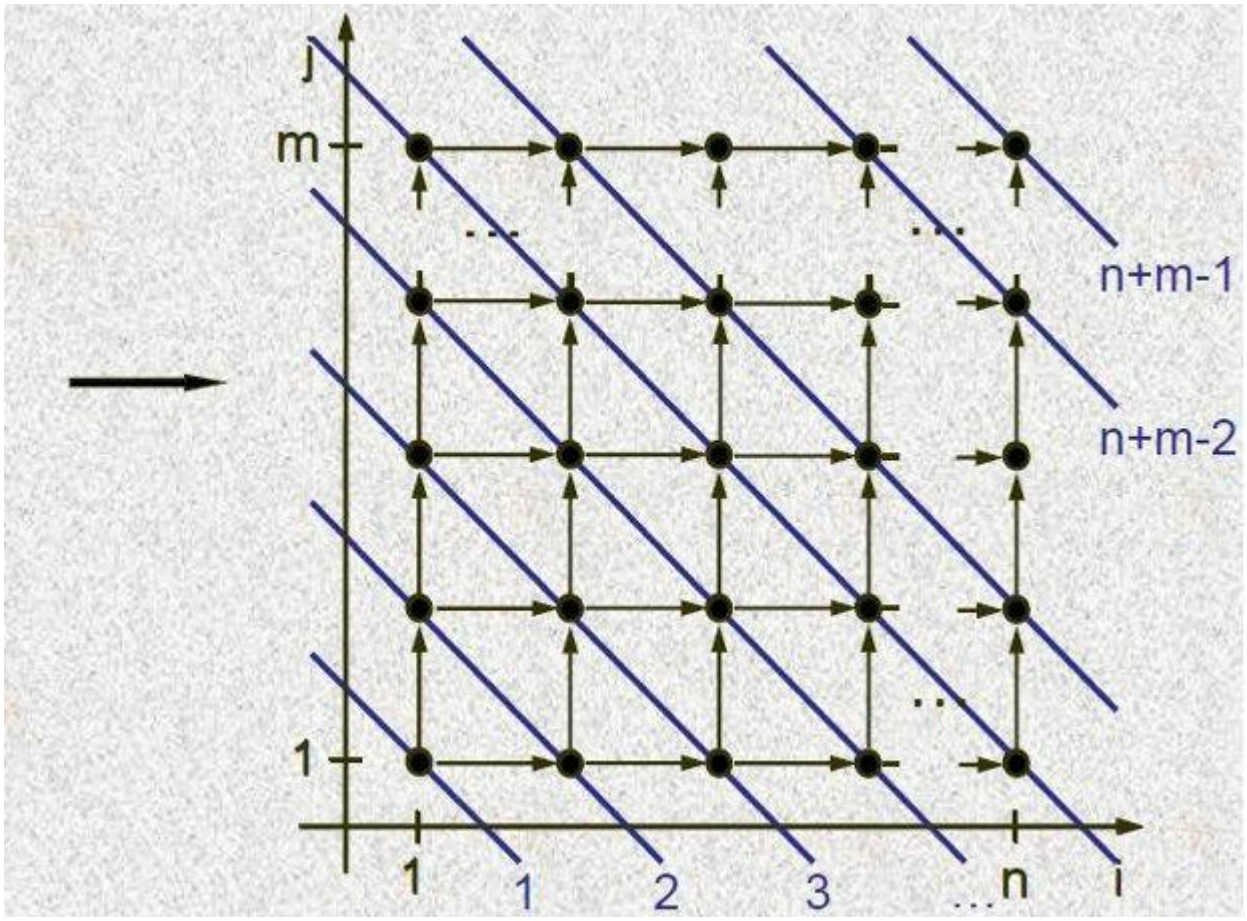
Δ_i — это многогранник из тройки, G_i = {f1 = i1,

i1 — это параметр анализируемого цикла,

f1 — это первая компонента векторной функции Fi из тройки.



Б) Скошенный



ризма

Виды параллелизма

конечный
(кол-во независимого ветвей одного вих при ∀ значениях входных данных)

(компы, опирающиеся на матем. команды)
суперскал. компьютеры

массовый
(опирается на независимость отдельных срабатываний итераций)

for (i=0, i<n)
a[i] = b[i] + c
(важен на практике)

координатный (лучше) **скошенный**
Различие.

Пример:

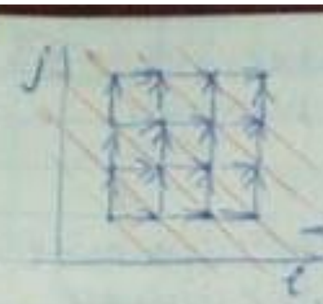
пометим опер-ор II-ным → for (i=0, i<n, ++i)
for (j=0; j<n, ++j)
a[i, j] = a[i-1, j] + b;

это урав. II-но по оси j компилятора

– информационная структура
последовательно

Это и есть координатный.

Но есть и другие ситуации
for (i=0, i<n ++i)
for (j=0 j<n ++j)
a[i, j] = a[i-1, j], a[i, j-1]



Вопрос: А есть ли вообще II-м здесь
Найдем критический путь: 2n-1
общее кол-во операций n2
Он есть. А ширина – n
Здесь всё что попало на 1 ярус – II но
ярусы послед-но таких ярусов 2n-1

За-да-ча! Такой II-м неудобен, т.к. кол-во вершин на каждом ярусе – разное => несбалансированность и ещё надо переделывать фрагмент
т.с. внешний – перебор ярусов, а внутренний – перебор вершин.

Это и есть **скошенный**.

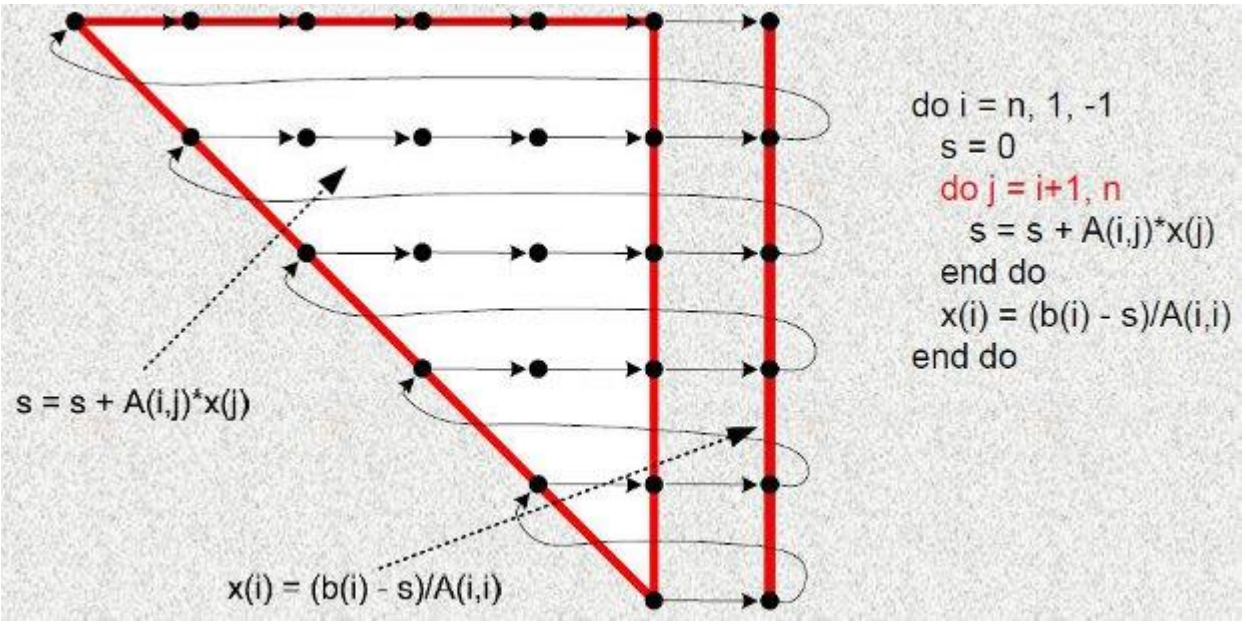
47. Ярусно-параллельная форма графа алгоритма, высота, ширина. Каноническая ЯПФ.

Граф, размеченный в соответствии с утверждением 4.1, называется ***строгой параллельной формой графа.*** Если в параллельной форме некоторая вершина помечена индексом *к,* 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 *к,* равна *к*—\. 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.* Слово "максимальная" здесь означает, что в этой параллельной форме в ярусах находится в определенном смысле максимальное число вершин. Все канонические параллельные формы являются максимальными.

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

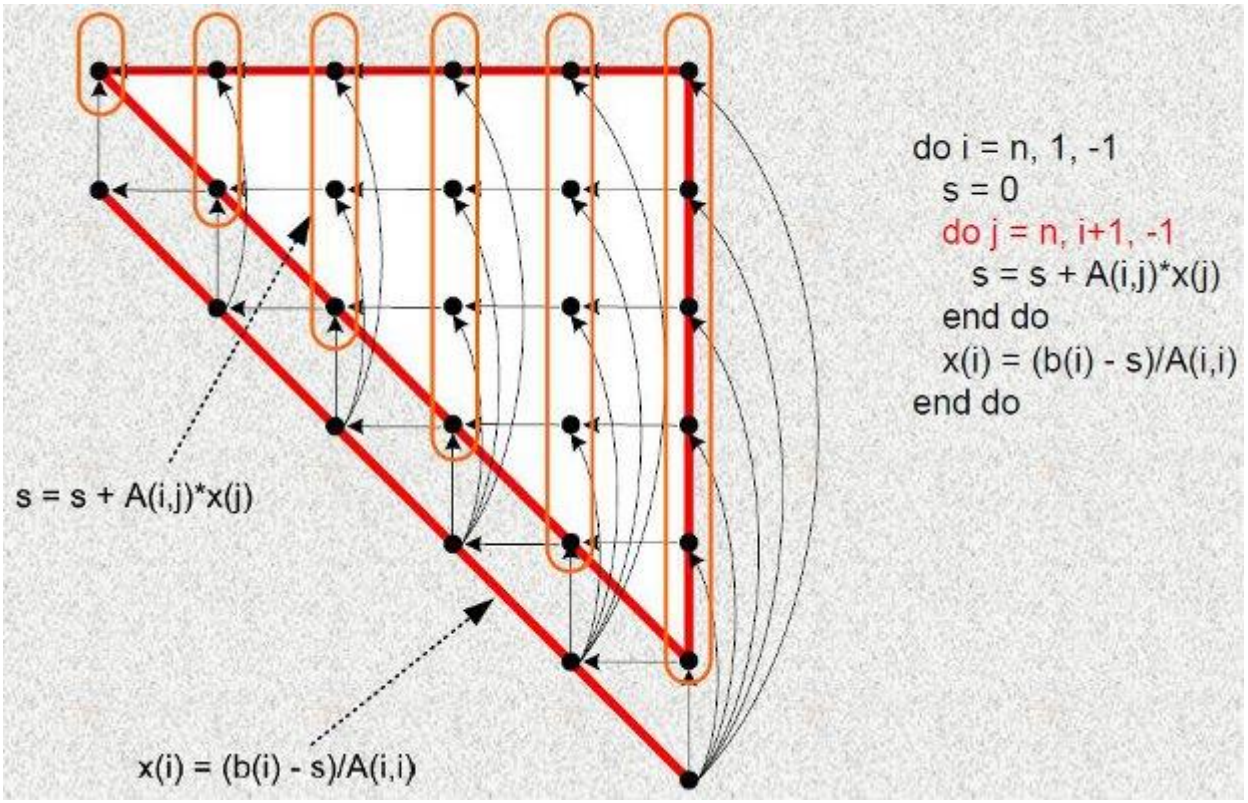
Рассмотрим решение верхней треугольной системы методом Гаусса.

Решение СЛАУ от метода к алгоритму (Информационная структура)



Критический путь графа алгоритма проходит через все вершины, следовательно, такую программу нет смысла исполнять на параллельной вычислительной системе!

С точки зрения метода Гаусса нет разницы в каком порядке выполнять итерации внутреннего цикла по j (суммирование). Изменим в программе только этот порядок и определим структуру.



Критический путь графа алгоритма имеет длину o(n), следовательно, данная программа обладает хорошим ресурсом параллелизма.